UiO : Department of Informatics
University of Oslo

An analytical and performance study of random topologies in an Openflow network

Mahsa Samavati
master thesis spring 2013

An analytical and performance study of random topologies in an Openflow network

Mahsa Samavati

22nd May 2013

An Analytical and Performance Study of Random graphs in an Openflow network

Abstract

Today's networks are getting more complex every day. Therefore studying connectivity, robustness and node importance features about them is getting more crucial. Since network technologies have strong relationship with network graphs and consequently with the concept of Graph theory in mathematics, an analytical model of calculation, comparison and prediction is established while experimenting prototypes of the model on a real platform.

The model creates a variety of both classic and random graphs via scripts and tries to use or define new connectivity, robustness and node importance variables on each type of the graphs. Later on, it will experiment the variables on a testbed with simulated network technologies.

The new technology, Openflow has been chosen as the platform to test the network technologies via a simulated environment. Despite traditional networks, this platform has a centralized controller as the brain of the whole network. The effects of that, would be considered in the experiments as well.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to express my special thanks of gratitude to the following people for their kind support during this master period:

- Ismail Hassan, as a wonderful person first, as well as a great professor and supervisor who inspired the first thoughts of the new platform in me and has been always there to support and encourage.

- Mark Burgess, as a unique person and a supportive supervisor who kindly accepted to be my supervisor and whose thoughtful advices brightened my way.

- Kyrre Begnum, as a knowledgeable professor and for his great workshops and precise comments.

- Hårek Haugerud, as a great person and a knowledgable professor who used to help students besides studies.

- Amiryasin Fallah Darya (Nima), as my best friend and classmate who has been so helpful all these two years.

- My family, my mother, father and brother for being so supportive while tolerating my absence.

- The love of my life, Øystein Andreassen for always being so encouraging and caring.

Chapter 1

Introduction

This chapter begins with motivation section where discusses about how mathematical graph theories relate to network topologies. It explains some effects of random graphs on today's traditional networks and brings up the questions of how these random graphs can affect Openflow as a modern network where the control plane is separated from the data plane in the whole network.

Further in problem statement section, the main research problems to support or enhance the topologies will be introduced and finally, thesis structure defines the structure of the next chapters respectively.

1.1 Motivation

1.1.1 Random graphs and traditional networks

Studying random graphs in traditional networks has led to surprisingly improved results [23]. Before mostly classis, small and geometric graphs were used in traditional networks. A variety of flat, star, tree, geometric, small world and even cubical graphs are among these categories.

In mathematics, graph theory developed increasingly by the concept of random graphs which is now connected to network technology world as well [48]. A typical random graph starts with n number of nodes and constructs edges between them at random. The difference between the random graphs is the probability distribution in each graph [4, 29].

Thus, one may ask *where is the intersection between these mathematical graph theories and networking topologies*? The answer could be that the

mathematical facts are pillars to lean on and make the rest of the technology on top of. In this case, the graph theory can lead to below mathematical issues [13, 25, 33, 68, 70] :

- Adjacency graph
- Adjacency matrix
- Node degree diagonal matrix
- Laplacian matrix
- Eigenvalues and eigenvectors
- Probability distributions

When the probably unfamiliar facts above come to networking business requirements, they can be translated to network topology, network neighbors, connectivity, robustness, growth pattern, etc.

Another intersection between graph theory in Mathematics and network topologies is the *possibility of pattern prediction in terms of network growth*. This can be a big scientific and business requirement issue that might be answered by the help of mathematical graph theory [11, 40]. The point is, practical issues such as multiple edge switch capacity which makes the theory possible in practice has to be considered carefully as it can limit the results in reality.

1.1.2 Software Defined Networking and Openflow with graph theory

Nowadays, there is a transition from the traditional networking to Software Defined Networking (SDN) and Openflow in particular which makes the control plane act separate from the the data plane. It is believed that this is the answer to the application based networking [34] and it is trying to find its way to address some scalability issues by having several physical controllers and one logical controller (Flowvisor technique [55]) or having several event based controllers (Hyperflow technique [61]).

Despite the variety of techniques that are being examined in the Openflow field, the study of network topologies and the reaction of such controller based networks to the graph theory, specially to the random graphs can be the key of developing network topologies faster and smoother in Openflow networks.

1.2 Problem Statement

In order to come up with Openflow enhanced models in terms of best topologies, a topology evaluation is taken as the main approach in this thesis. First an analytical graph investigation is done by probing mathematical features of well-known network graphs(classic and random graphs in particular). Graphs are scaled up in each test and performance factors are considered numerically among overall tests. Finally some practical tests are taken to prove the results and the range of influence in an Openflow network. Therefore, this thesis tries to address the research problems as below:

1. What features can be used as variables to measure connectivity, robustness, spreading and centrality of a classic and random network graph?

2. What is the behavior of network topologies in case of connectivity, robustness, spreading and centrality variables?

3. What can be a prediction and categorization for the classic and random network graphs while scaling up?

4. How real world and Openflow perform within different network topologies?

1.3 Thesis Structure

The structure of this thesis begins with Background as the following chapter. Since the topic in practical part (Openflow) is new, the literature has been gathered considering a variety of aspects of the technology. Network graphs and Openflow controllers that are going to be examined further, have been introduced in more details.

Chapter 3 (Approach) gives different specifications that make the whole methodology. The corresponding approaches to each specification have been defined thoroughly in this chapter.

The following chapter 4 (Results and Analysis) shows the actual and analytical results as well as analyzing each topology growth to come up with the best models. Probing the possibility of using a prediction model while scaling up a graph and general predictions with the category of random graphs in particular are taking place with the analysis chapter as well. A number of tests in an Openflow environment are done to examine the findings within the new technology.

In chapter 5 a Discussion of the previous chapter with a separate categorization of each variable that was defined in the analysis is given with an overall discussion of the whole classic and random topologies behaviour.

Last chapter (chapter 6) with the Future work and Conclusion tries to give ways to continue the research as well as addressing the research questions in the problem statement section precisely. Each answer would be mapped to corresponding results and analysis.

Finally, Appendix chapter covers the most significant scripts that have been used in the previous chapters.

Chapter 2

Background

This chapter is going to study literature survey of Graph Theory and Random Graph Network as the main theoretical part of this thesis following by Software Defined Networking (SDN), Openflow as the main protocol based on SDN and related terms in this area as the main practical and testbed of the thesis. Additionally some of the examined graphs and topologies as well as the tools that are going to be used will be introduced thoroughly.

2.1 Graphs

2.1.1 Graph Theory

This term belongs to both mathematics and computer science area. Theoretically a graph consists of two main features; vertices and edges. Vertices are mostly called nodes in computer science or mainly in computer networks. Edges are the connections between the nodes [4].

A graph can be directed or undirected in terms of direction from one node to another. An example of a directed graph can be a web site. This thesis considers undirected graphs as it probes computer networks at switch layer. Thus, nodes are most likely (switches/ servers).

2.1.2 Adjacency Matrix

This is an nxn matrix where n is the number of nodes in the network. Considering an undirected graph this matrix consists on 1s and 0s. 1s

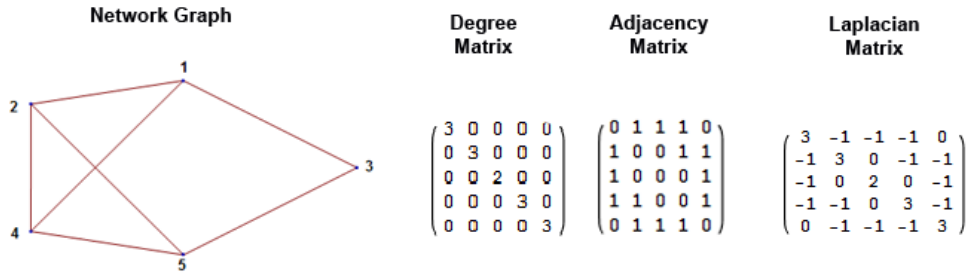

Figure 2.1: A five node Graph and corresponding matrices

happen when an edge exists and 0s happen otherwise [33]. Adjacency matrix is the basis for some further calculations in this area since it is unique for each graph.

2.1.3 Node degree and Degree Matrix

Node degree is the number of edges that meet each node [25]. Converting this information to a matrix, it is called a degree matrix which is a diagonal matrix representing the degree for each node. In combination with adjacency matrix, this is used to make Laplacian matrix of a network graph (see Figure 2.1).

2.1.4 Laplacian Matrix

This is simply defined as below where D and A are Degree and Adjacency matrices respectively:

$$Laplace = D - A \tag{2.1}$$

Thus, Laplacian matrix contains information about both the matrices above [70]. The Laplacian matrix is also the basis for further numerical analysis in this thesis.

Figure 2.1 shows an example of a simple undirected 5 nodes graph and its corresponding matrices.

2.1.5 Eigenvectors and Eigenvalues

The Eigenvector of an nxn matrix A is shown by V. When V is multiplied by the matrix A, the result will be A by a number (lambda) which is called Eigenvalue. The formula is shown as below [69]:

$$Av = \lambda v \tag{2.2}$$

Eigenvalue shows the length and direction of the eigenvector. In practice, Eigenvalues are used in ranking nodes in large networks. As largest Eigenvalues show the most important nodes in terms of number of connections. Respectively, the most important Eigenvector shows the centrality of the nodes which means the influence of them in the whole network [68].

Eigenvalues of the Laplacian matrix determine some other features of the network such as number of connected components and rate of convergence [70].

2.1.6 Epidemic spreading

Epidemic spreading is about how critical a node is to pass information through, as smooth as possible. This is usually considered by random walks and how often the node is accessed randomly. This is related to eigenvalue as well and is measured by maximum eigenvalue if adjacency matrix of a graph (network topology [67]. This would be discussed more further in the next chapters.

2.1.7 Algebraic connectivity

Algebraic connectivity is about how well connected graph G(V,E) is. This is measured by by second minimum laplacian eigenvalue and the graph is considered connected if algebraic connectivity is greater than zero [10].

2.1.8 Betweenness Centrality

The term node importance of a network graph may refer to a way of measuring all shortest paths that pass through the node. This also means all the load that the node can take which is called Betweenness Centrality. The formula is given as below [16] which shows the fraction of the shortest

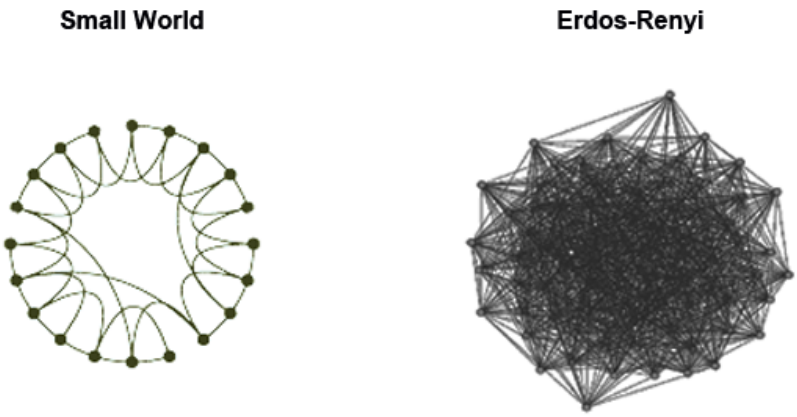

Figure 2.2: Small world versus Erdos-Renyi random graphs

paths from s to t via v to the shortest paths from s to t. The condition for this formula is that non of the s and t and v are the same nodes:

$$BC(v) = \Sigma \frac{\sigma_{s,t}(v)}{\sigma_{s,t}} \tag{2.3}$$

2.1.9 Random Graphs and network topologies

A random graph is created by n fixed nodes whose edges are added randomly. Thus, Random Graphs as a theory talks about both graph theory and probability.

Speaking about best network topologies, the questions is *how many edges should connect network nodes and where should they located*?

Erdos-Renyi model is an example of random graph theory where each edge is constructed with the probability of P which is totally independent of other edges.

Small world is another example of a random graph where most of the nodes are tried not to be neighbors but still accessible via a small number of other nodes. Figure 2.2 shows the structure of the above graphs.

2.2 Traditional Networks

Current network devices have very specific hardware which runs a specialized Operating System (OS) with some general or specified operating features. This is interpreted as a Close Environment, as configuring such a network is done box by box (device by device) and all the data plane, control plane and management plane is within the same device [35, 38, 61]. Devices are often very complex, heavy and expensive while a thorough network administration is very hard to achieve.

Consequently, this does not let any interaction with upper layers, especially application layer, which means the current approach is not compatible with business requirements as well.

2.3 Software Defined Networking (SDN)

Software Defined Networking breaks through such closed network topology to the application layer world, by the help of well-defined open APIs (Application Programming Interface). This means experts with the ability of software programming can write an application/feature that would be run on a Network Operating System (Network OS) or simply network software [24].

The aggregation of the terms above (API, Network OS) besides a Network Device (Hardware), is what builds the structure of Software Defined Network. In this approach the Hardware is not a very specialized one. It can be just a packet forwarder which is not hosting the APIs and Network OS. The Network OS is what is called the Controller and is nothing but a piece of specialized software. There should also be a Common Protocol to communicate between the controller and the actual physical/virtual hardware. The general infrastructure is shown in Figure 2.3.

Each of the hardware in this network has one piece of built in control software to communicate with the main controller [44]. The applications can be among a variety of software such as Firewalls, Vlans, Load balancers, etc. This will bring a great opportunity to software business in order to program the network from layer 7 which is the application layer.

Consequently to give a final definition of SDN, one can say that Software Defined Networking is a networking approach that has the control plane detached from hardware and accumulated with a software controller.

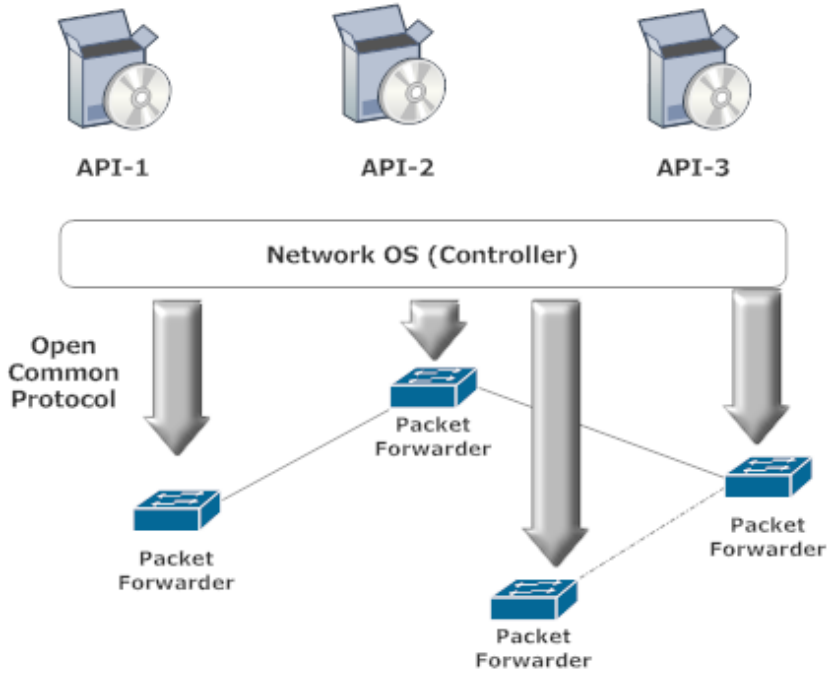

Figure 2.3: Infrastructure of Software Defined Networking

2.4 SDN controller (orchestrator)

SDN controller which is also called as SDN Orchestration is mostly responsible to respond to controlling APIs. This means there must be a common communication between API software and the controller [36]. Thus, the APIs use an object model (abstraction interface) to describe themselves and the orchestrator manipulates them. The point with this is that the orchestrator is physically single, but logically acts distributed among different APIs.

2.4.1 Application interface

What described above is also called application interface. In other words, this interface lets the orchestrator to communicate with the applications. This is also called North Bound interface, as it refers to north-bound of the SDN infrastructure.

Generally, the application interface enables applications to do the following [59]:

- Bootstrapping

- Authentication
- Learning
- Interaction with the controller

2.4.2 Plug-in interface

Additionally, there must be a level of abstraction interface in which the orchestrator communicates with the control plane in the network device. This is done via what is called SDN Plug-in. A plug-in interface makes the device talk to the controller and exchange information via negotiation [59].

2.4.3 Policy interface

Policy interface enables the orchestrator to communicate with different databases in the infrastructure [32]. The main databases within this region are authentication, authorization and policy databases.

2.4.4 Orchestrator interface

How controllers communicate and interact with each other is done within orchestrator interface [59]. This context can be used in scalability architectures by the help of a protocol such as Openflow.

2.5 Openflow

Openflow is a new standard in the world of SDN which gives more power to the hands of network administrators by introducing an open common protocol [14].

Supposing an enterprise data center with thousands of physical and virtual servers connecting to the network, grouping Vlans on a device by device basis is unbelievably difficult and with dynamic changes added, it would definitely turn to a real disaster. With the help of SDN in concept and Openflow as a technology, this has become quite straight forward as controller software can program Vlans and track the whole network with Openflow standard.

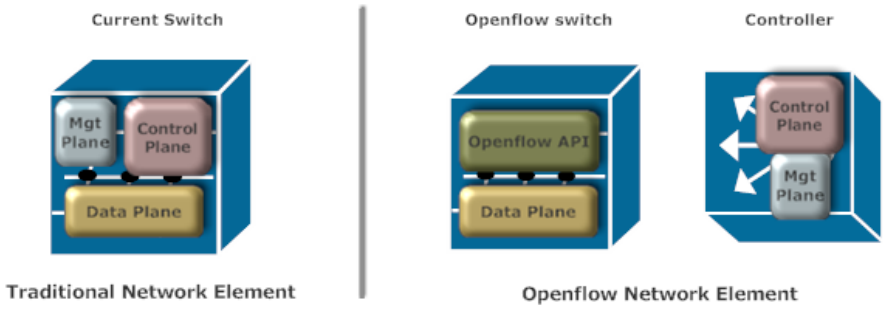

Figure 2.4: a. Traditional Network Element - b. Openflow

In order to have a better understanding of Openflow as a network element and its relationship with SDN controller, a schematic study of a traditional network element in contrast with Openflow network element would be quite helpful.

As seen in Figure 2.4.a, a traditional network element is a complex device which has all required features inside [14, 42]. These features are mainly Data plane, Control plane and Management plane, which are built in among today's network devices.

In contrast with traditional devices, an Openflow network element, besides having the same Data plane, has an Openflow API which is in deep contact with separate Control plane and Management Plane that are now available on SDN controller [24, 43] (See Figure 2.4.b).

The way Openflow API communicates with the controller is via what is called Openflow protocol to investigate how to move the traffic [24].

Merging the SDN idea with Openflow protocol, means to take the brain of the whole network out to the controller. In such an approach, Openflow can be interpreted as standard signals which talk between the controller (brain) and the hardware (body).

2.5.1 Simplified Openflow Mechanism

Supposing the network in Figure 2.5 as a traditional network with no separated controller, there has to be separate configuration on each switch for the proper Vlans for instance. Current switches will also take care of loops by running Spanning Tree Protocol (STP) or some routing algorithms.

With Openflow approach, there would be no port Vlanning and the controller defines the Application Flow which simply says which application

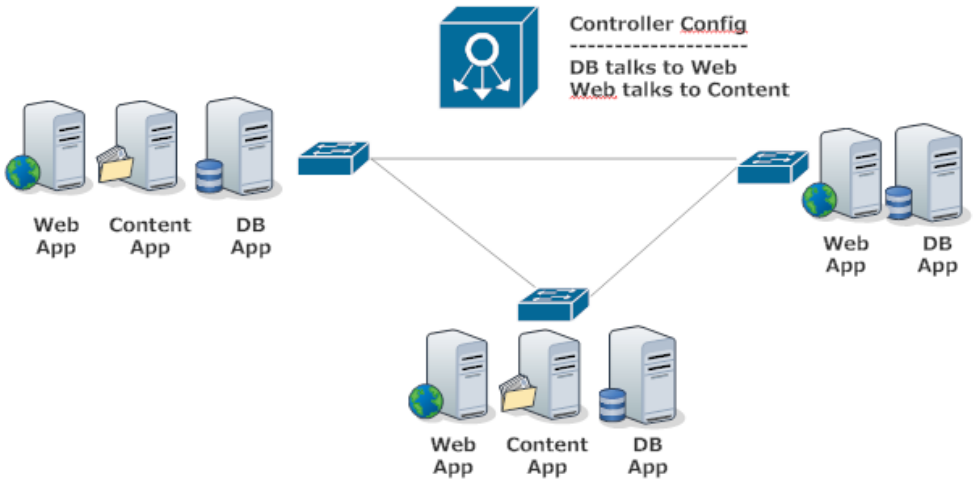

Figure 2.5: Openflow Mechanism

can talk to the other. These policies will be pushed securely to flow tables of all switches by the controller [41].

In this case if an application moves, nothing bothers the whole network simply because the network has been programmed by applications from controller view. Another fantastic idea is that the applications themselves can talk to the controller and inform it with unavailability, changes and capabilities. In the same way, no routing protocol will be set by the switches as well.

The mechanism above brings more flavors to the world of network administration by the cases below:

- Scalability
- Flexibility
- Easy administration
- Robustness

2.5.2 Northbound Southbound model

New network technologies such as Transparent Interconnection of Lots of Links (TRILL) introduce a new term which is called East/West traffic flows. The idea has also come from network virtualization where in large data centers the traffic flow might travel long ways across the whole network which is spread along a wide area from right to left (east to west). This

implies a problem in this architecture where core switches are also spread along the network in an east/west manner. These networks are hard to update, manage and have great latency.

In the same architecture, there is a north/south point of view which implies the flow of traffic from WAN cores to the edges and from the edges to servers and clients respectively [20,39].

Recently, another northbound southbound discussion has come up in the world of SDN and Openflow. The terms imply north and southbound APIs, which are basically controller applications as northbound and Openflow device APIs as southbound APIs. Precisely, SDN controller to Openflow device calls are considered as southbound and SDN controller to applications negotiation is called northbound [20].

An example of southbound API is NETCONF which has played a good role in combination with Openflow to enable network configuration management. Another example is SNMP (Simple Network Management Protocol) which is not quite combined with Openflow devices but is still playing the same role as it had in traditional networks to manage the whole network by Openflow's support for SNMP [73,75].

On the other hand, there is a variety of northbound applications, such as different firewalls, load balancers and topology discoveries. Applications in this area are changing rapidly as software world changes and there is no standard applied to them.

2.5.3 Network Flows

Network flows are basically packets. But these packets have some common characteristics which make them appear as a flow. The main characteristics are as follow [8,54]:

- Same source and destination address
- Same source and destination port
- Same protocol
- Same period of time

In case of UDP protocol which is a uni-directional protocol, packets with the above features make a single flow each time. On the other hand, with bi-directional TCP protocol, mentioned packets make 2 flows each time.

As an example, a flow can be a point-to point communication. Since the communication happens within a specific amount of time, there are several connections in reality. That is the reason why these flows are called Virtual Connections. The point with the flow networking is that, they can be treated as queues in switches, to shape the traffic, load balancing or in more general term, Quality Of Service (QOS) [21].

Openflow is a flow based protocol, which brings a high performance by grouping up packets and use them as a flow entry to get pushed to its flow table.

2.5.4 Openflow Flow Table

A Flow Table is an entity within an Openflow switch (a switch might have more than one flow table). Every flow table consists of flow entries which are basically match fields, with priority and some actions [49,71].

If an entry for a new packet does not exist in the Openflow flow table, the Openflow switch will send it to the controller to make the final decision for that [17,46]. If the controller does not decide to drop the packet, it will add it up to the flow table as a new entry. From now on, the same packets would be forwarded in a similar manner within this entry.

2.5.5 Network Virtualization and Openflow

Network virtualization is the combination of hardware and software (as it is in common virtualization), plus network facilities that are aggregated into one software management environment. Thus, network virtualization environment lets applications, features and end-point host/user to communicate within a single platform [26].

The terms software management environment, application communication and single platform, make network virtualization more familiar and closer to the context of SDN and subsequently Openflow as the standard. In a controller-based network infrastructure, network features are aggregated into the controller which makes all the routing and forwarding decisions and push them to the flow table in Openflow switches.

In the same approach, Openflow can take the advantage of network flows and group them to separate logical slices of network, so that it will be possible to create a virtual flow network on top of the physical network testbed [5].

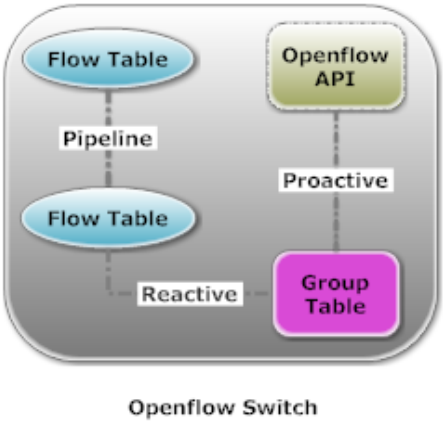

Figure 2.6: Openflow Switch

2.6 Openflow switch

An Openflow switch can contain more than one flow table. Besides it has a group table which is responsible to do the forwarding. Openflow API talks to the main controller with Openflow language as the standard protocol [28,56].

What the controller can change in the flow table by pushing flow entries is add, delete or update. These actions can be taken proactively by beforehand pushes or reactively as a packet response right away.

Each entry in a flow table contains matches, priority counters and actions to the matches. Matches are done in combination with priorities. If a packet needs further processing through other flow tables, it will be sent to other flow tables by what is called a Pipeline. This action is also called pipeline processing and it continues till there is not Next Table in the matching flow [7,18]. This is necessary for the packet to find the right match and be forwarded. Figure 2.6 depicts the internal actions of an Openflow switch and the relationship between them.

2.7 Openflow controllers

As discussed earlier and as a general look at an Openflow controller, the following characteristics can be given to define what such controller is [64]:

- Application-based
- Full connectivity with network devices

- Capable of discovering network topology
- Algorithm-based
- Capable of updating flow tables using Openflow

After discussing about the main roles of a controller in this part two of the most famous Openflow controllers will be introduced with some of their main features that have made them pioneer. At least one of them is going to be used mainly further in this thesis.

2.7.1 Floodlight controller

Floodlight is an open controller that has been written in Java which makes it easy to be run on a variety of platforms. It uses Apache as a friend of all network and system administrators and is meant to be used in enterprise level. At the same time it allows the network administrator to have a hybrid network of Openflow and non-Openflow based islands. The point with Floodlight is that it uses what is called REST API that makes it easy to run the commands as brief as possible. That is because the REST API defines commands and functions to send requests and receive answers via HTTP. Using HTTP makes the REST easy to run and test [64, 65].

After the short introduction, the main features of Floodlight are given below in brief:

- Java-based
- Apache-based
- Module based (makes it easy to extend)
- Open source
- Less dependency (easy installation)
- Broad support of Openflow switches
- Proactive network management with the static flow pusher API

The last feature belongs to one of Floodlight's applications. Below is a list of Floodlight's applications with a short description for each of them:

Static Flow Pusher: This application enables the administrator to manually add (push) flows to the network.

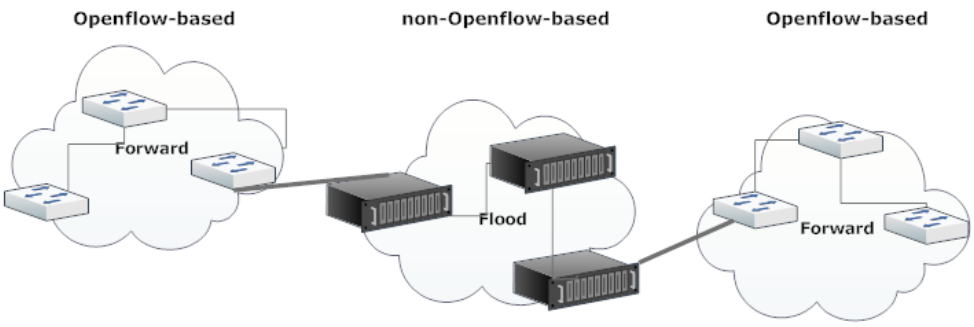

Figure 2.7: Hybrid Network

Circuit Pusher: This is another REST API which performs IP-based permanent flow entries to all the devices. This can also be interpreted as a bidirectional circuit in an Openflow network.

Firewall: This application is a Floodlight module based on ACLs (Access Control List) which permit or deny ingress flows to the Openflow switch. This firewall supports wildcards in the network as well.

Virtual Network Filter: This is a module which makes MAC based L2 virtualization possible. It basically means that you can have multiple L2 virtual networks within the same L2 domain.

Forwarding: Forwarding module has the responsibility to forward flows from one device to another within the network. This might get more complicated in hybrid networks containing both Openflow and non-Openflow islands. The mechanism works in such infrastructure although it might not be optimum. The Forwarding module would flood to packets/flows when it faces the islands with no Openflow device (Figure 2.7).

Openstack Quantom Plug-in: Openstack is a software for making clouds (private or public) and Floodlight can make the testbed network for Openstack by the help of an Openstack plug-in which is called Quantom. Quantom supplies Networking as a service (Naas) and Floodlight connects to Quantom in order to work with Openstack.

2.7.2 Pox Controller

Pox is another Openflow controller which is written in Python language. It can be run under a variety of platforms such as Linux, Mac and Windows and it is quite straight forward to install and use. Pox is module-based and the modules are sometimes called components. Below are some of the main

components of Pox with a brief definition [6, 64]:

forwarding.l2_learning: This is a component which enables the Openflow switch to become a layer 2 learning switch.

forwarding.l3_learning: Unlike the name this component is not a complete router, but is used to test ARP requests and responses.

openflow.discovery: This component sends out special messages of Link Layer Discovery Protocol (LLDP) to discover network topology.

openflow.spanning_tree: This component takes place after knowing about the topology of the network and provides a Spanning Tree to have a loop free and more robust network.

openflow.keepalive: Some Openflow switches might consider an idle connection as a dead connection. This component makes sure to send echo messages periodically to prevent connection loss.

misc.dns_spy: This component does monitoring to DNS responses and stores them so that other components can have access them whenever it is needed.

Besides the above components, Pox consists of some events and actions that are meant to be used with Openflow protocols. Events such as Connection up/down, Port status, FlowRemoved and actions such as Set Vlan ID/Priority, Set Ethernet Src/Dst address and Set IP Type of Service are some examples to give an idea of what Pox is capable of.

2.8 SDN and Openflow Funnel

All the background and definitions above lead to a funnel model of SDN and Openflow in the form of AS-IS which identifies both Openflow and non-Openflow tools and regulations that are being used today in an Openflow network.

As it is shown in Figure 2.8, the model is presented in as a funnel with different layers of abstraction to show how different parts cooperate with each other [19]. As it is seen, the lowest layer is the hardware where network devices take place. Although this can be network virtualization layer as well, the term hardware is still common to use. Thus, the devices are basically Openflow switches that communicate with upper layers by the Openflow common protocol.

Although Openflow has some of its own built-in management functions,

the impact of old management protocols such as NETCONF and SNMP is inevitable as they still have their place in an Openflow network as well as in a traditional network. Management protocols have communications with both upper and lower layers that are controller and Openflow protocol and devices.

The next layer up is where controller lies. There are some proprietary controllers growing in the market as different brands are trying to go with the Openflow river. Depending on the market, next generation controllers might be in one of the places below [27]:

- At the same orchestration area
- Integrated with network device
- At several places consisting of the platforms above plus other platforms

The application layer is where different APIs take place. Variety of these north-bound APIs is a lot but there are still some main applications that have to be implemented to have a secure scalable Openflow network that can solve the traditional network problems thoroughly. The market is still not rested in this area, but the need for basic standardization of these applications is felt within the technology.

Finally, user interface tools and standards are being used to communicate with different parts of the system as it was in traditional networks as well. Although they are depicted on top layer but the impact of them is spread through the entire funnel as this is how a user/administrator can have access to the whole system at different levels.

The reason a funnel is depicted in Figure 2.8, is that the wider the funnel gets the more variety of tools and software can be named and the more they can change and affect the whole system as well.

2.9 Openflow development models

To begin with, Openflow turned to be a simplified model to remove complexity in traditional networks. Since it was used in research areas, it kept simple for a while, but to match the market's requirements, it has been developing recently.

Thus, this part is going to introduce some of the main deployments of Openflow:

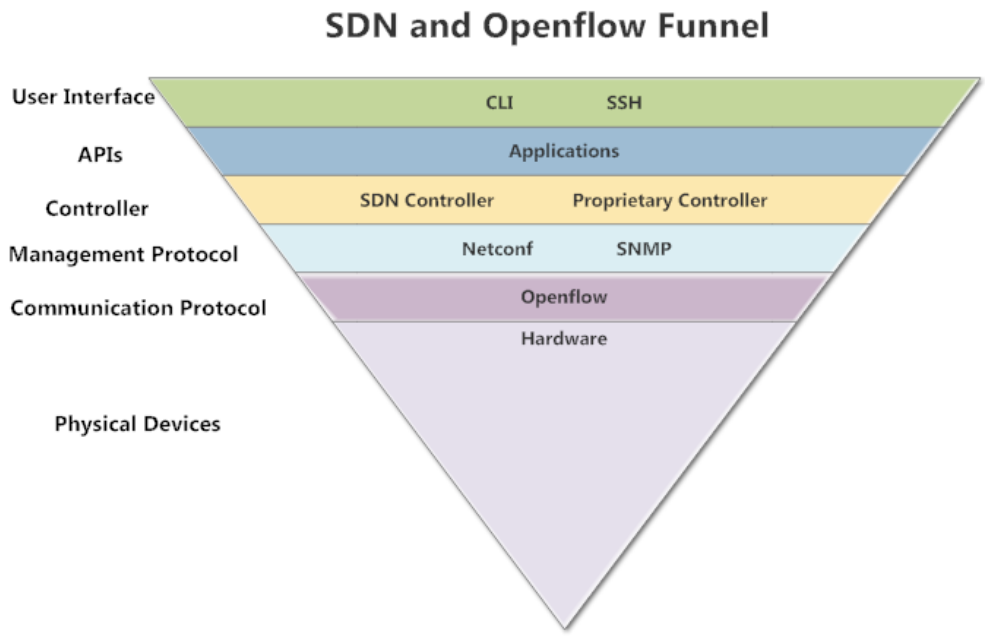

Figure 2.8: Openflow Funnel

2.9.1 Primitive Openflow

This has come from the very first idea of a simplified Openflow in which, the switches are thoroughly dumb [37]. The idea supports the controller to be the most functional element of the network, including all the control-plane responsibilities. Below are the main disadvantages of such model:

- Placing the entire load on the controller sounds unreasonable especially with a growing network.
- The connectivity of all switches to the controller is required.
- The response time would decrease because responses come from a central controller.
- Loops should be avoided with an STP (Spanning Tree Protocol) mechanism [58].

2.9.2 Primitive Openflow with expansions

This model assumes a switch to have a separate low-level control functionality. This small control-plane can take the responsibility of running low-level topology discovery and linking the aggregation groups. An example

of extension to Openflow is Openflow 1.1 support for multi-pathing which enables low level load balancing [66].

2.9.3 Ships in the night

Switches in this model have a control plane which is close to traditional one. They refer to the controller in case of certain VLAN settings, trunk links or some other specific ports. This localized control-plane sends out link status messages to the controller [63].

The problem of direct connection of switches to the controller has been solved by running TCP sessions in between. This approach is being used in hybrid environments with Openflow and non-Openflow networks.

2.9.4 Integrated Openflow

This model is the results of the context that is called Programmable Networking. The switches are now functioning as routers with separate data and control plane, which is also connected to the main controller at higher level. In this model, Openflow static routes have the ability to redistribute to common routing protocols [22,72].

In order to allow the controller communicate with lower level routing protocols, routing tables are designed inside Openflow controller. This model can be a successful example of combining traditional networks with Openflow at the edge to take the advantage of Openflow flexibility and programmability in such network.

This method also uses the traditional functionalities that are hard to implement, but gets far from the initiative SDN and Openflow promises. In order to have such combined technologies, some special purpose northbound APIs and protocols must be present as well.

2.10 Ships in the night versus Integrated

Although ships in the night model has a level of integration to some extent, it is not fully integrated. The reason for that is that the Openflow switch in this model has some ports that are controlled by Openflow and some that are under the control of traditional switching and routing approaches. In the integrated model this separation has been removed and no partitioning is applied to the device's resources so that the traditional approaches are fully integrated with Openflow protocol in this model [72].

2.11 Programmable networking

Openflow by itself is only a protocol, and what makes it more interesting is the ability to program the network. Thus, the things are not decided all by network devices. Decisions are made by software which tells the device what to do. But this does not end here. There are more issues to consider as there is not a good connection between applications (APIs) on top-level and low-level network devices [22, 50].

Therefore what is basically called network programmability is the right interaction between networking world and application world. APIs take the information they need from the network and on the other hand, network makes better decisions by the help of applications. Below are some the main advantages of having a programmable network [12, 53]:

Real time corrections: This occurs when a bi-directional knowledge is shared between network and application.

Clear network control: By the help of network hardware and algorithms, it is possible for the applications to have a more effective control and network forwarding would be more flexible by the information from the applications.

Better data utilization: Having a clear network overview by the cooperation above, the amount of data in the network being utilized by applications increases rapidly.

Better development: The world of applications seems endless. This can change networking world in the same manner by opening multiple doors of opportunity both for the science and the market.

2.12 Some network graphs

In this section the graphs that are going to be examined mathematically in this thesis are introduced briefly. [15]

2.12.1 Classic graphs

Balanced Tree is a tree which is perfectly balanced. It has a height in depth and a branching in width as its main characteristics.

Ladder is a ladder shaped graph in a row of two nodes (pairs). Each of the

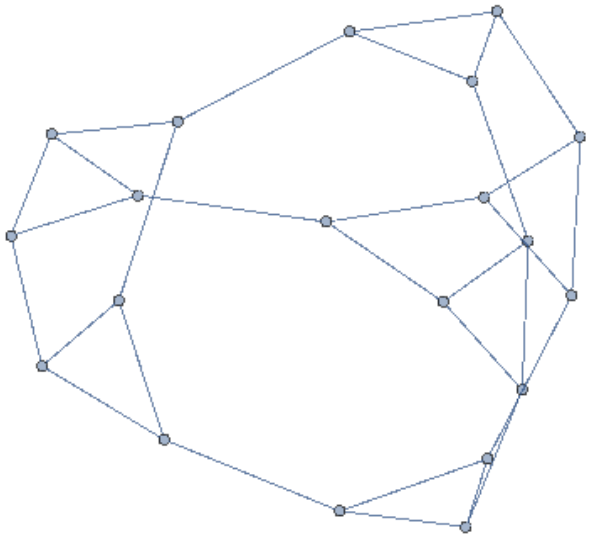

Figure 2.9: Random Regular Graph with 20 nodes and degree of 3.

pairs are connected by one edge.

Star is a star shaped graph with n nodes connecting to one central node. The total number of nodes would be *n+1*.

2.12.2 Random graphs

Fast GNP is a binomial Erdos-Renyi graph with P probability which chooses *n(n-1)/2* possible edges where *n* is total number of nodes. It behaves faster than Erdos-Renyi for small Ps.

Powerlaw Cluster is an algorithm which represents graphs growing with power degree distribution.

Watts Strogatz is basically a small world graph which constructs a ring over all nodes and each node will be connected to *k* neighbors with *P* as the probability.

Random Ragular is a graph with no parallel edges. The feature *D* defines the degree of number of edges that are constructed. Figure 2.9 shows a random regular graph of 20 nodes with degree of 3.

Barabasi Albert is a primitive graph with *n* nodes and no edges which starts attaching nodes with *m* edges to each other with high degrees.

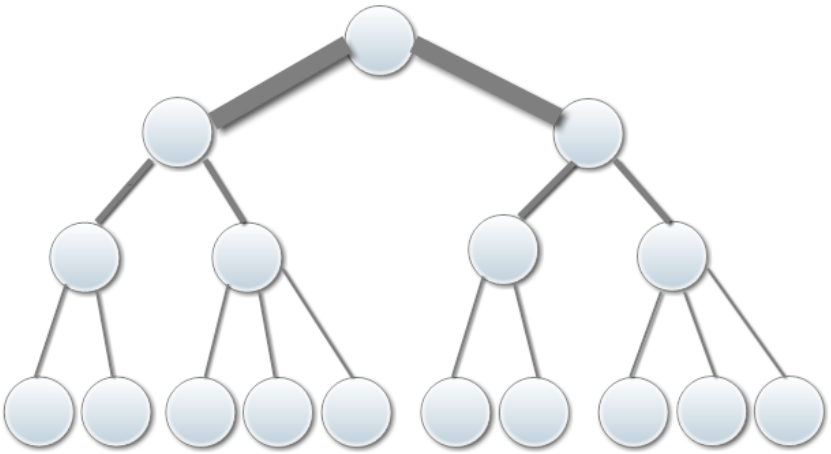

Figure 2.10: Fat Tree Topology

2.13 Some networking topologies

2.13.1 Tree topology

A tree topology is a mix of two or more ordinary star networks. Each star has a central switch with several hosts/servers connected to it [62,76]. The tree topology is seen in group networks within a small physical place. For instance a company with different departments or a university network can have a tree topology.

2.13.2 Flat topology

Flat networks are getting more popular as an alternative to tiered topology networks. They are meant to increase network performance especially in terms of bandwidth as they eliminate STP which can be a barrier to use all available paths [51].

2.13.3 Fat tree topology

Fat tree is in contrast with ordinary tree topology which is now considered as a thin topology. Fat tree is a tree with fatter links on top (close to the root) and thinner ones at the bottom (see Figure 2.10). This can bring more efficiency in terms of bandwidth and can also be a suiting concept to the world of scalable networks [52]. Fat tree is easy to make and very cost effective as well.

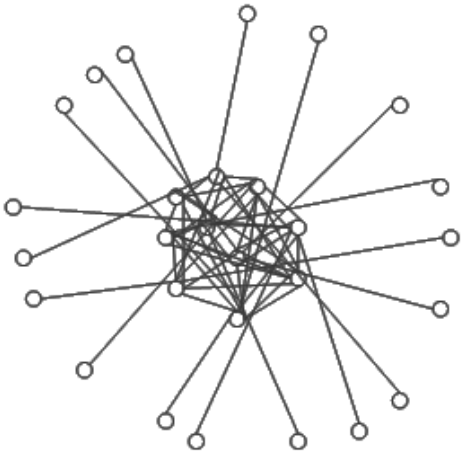

Figure 2.11: Jellyfish Topology

2.13.4 Jellyfish topology

Jellyfish is meant to be flexible as it creates a random topology within the switch layer with offering equivalent or sometimes more bandwidth than the fat tree topology (see Figure 2.11). The way jellyfish is made is by joining a number of network pair switches randomly. The join is at the free ports and it continues to be finished with all free ports [57].

Jellyfish shortens the flow paths in comparison with fat tree which normally has longer flow paths. It assures this by making the hosts to be extended more evenly in the entire network.

2.14 Testbeds

2.14.1 Mininet

Mininet is a tool/software that makes it possible to have a virtual network instantly on common personal computers or laptops. The network is virtual but realistic and it is run on a real kernel. Mininet has a CLI to interact with the network devices that makes it brilliant to development and research activities [31].

Below are main goals of Mininet:

- Simple network platform

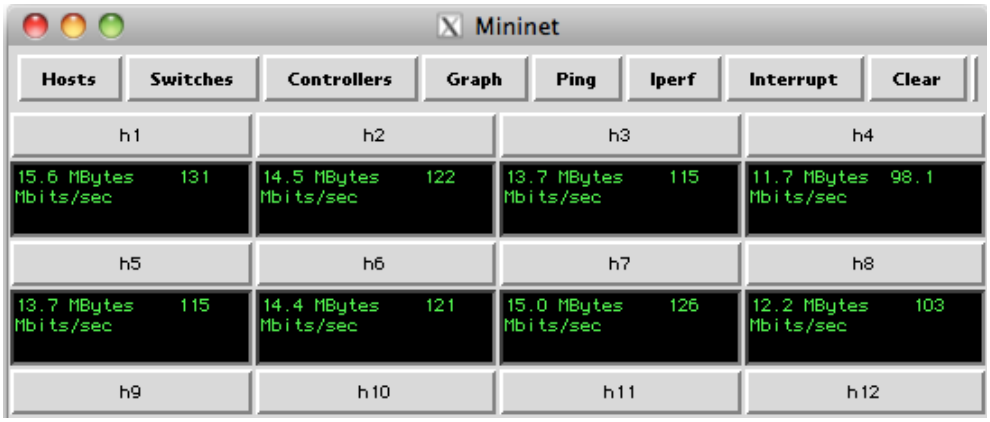

Figure 2.12: Mininet

- Support for different topologies
- A CLI integrated with Openflow
- Support for Python to create network topologies

Mininet makes a level of process abstraction on top of the Operating System and provides process virtualization and boots up several hosts and switches on one kernel (see Figure 2.12). Comparing Mininet with a thorough virtualization system, advantages below would show off:

- Less resources
- Fast boot-up
- Better scalability
- Easy installation
- Openflow compatibility

Mininet is going to be widely used within this thesis as a testbed for running different topologies and examining different Openflow controllers [60].

2.14.2 Networkx

In order to study complicated and random graphs with the ability to scale up the network easily, Networkx is used in combination with Mininet as the main testbed [9, 45]. Networkx is Python-based and has a library of different network graphs. Firgure 2.13 shows a network that is simulated

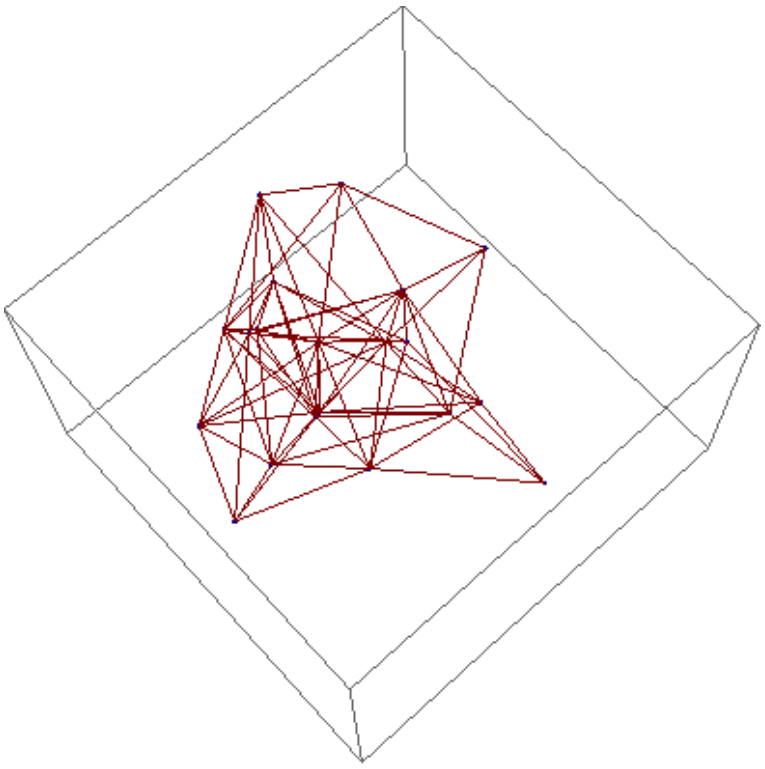

Figure 2.13: Networkx simulated network(graph)

by Networkx on Mininet. Below is a list of networkx main features as well:

- Graph diversity
- Random graph creation
- scalable networks
- conversion features
- Mathematical views (Adjacency,Node degree list, etc)

2.15 Testing and Benchmarking tools

This part introduces a couple of testing and benchmarking tools which can be used examining a variety of random graphs as well as Openflow features in Openflow controllers for the theoretical and practical part of the thesis.

2.15.1 Mathematica

Random network graphs that are constructed by Networkx on Mininet, are going to be numerically analyzed by Mathematica software. Mathematica is not only a scientific and mathematical program, but also an engineering software. Random networks are based on random graphs and Graph Theory that can be calculated analytically in Mathematica to gain better technical results [2]. Below are the features of Mathematica that are going to be used in networking and Graph theory(mainly in this thesis):

- Primitive and special math libraries
- Matrix tools
- 2D and 3D visualization tools
- Continuous and discrete calculations
- Statistics libraries
- Import and export filters for data and images
- Programming support

2.15.2 Cbench

Cbench or Controller Benchmark is a utility to examine Openflow controllers. Cbench assumes a number of given switches that are connected to the controller and sends out messages and watches flows being pushed to the controller. The messages are called packet-in and the flows being watched are called flow-mod.

Cbench benchmarks maximum number of packet-in messages and maximum port status messages as well. It also computes the processing delay at each stage [30]. The following Cbench command is a good practical example of this utility following with the options definitions in Table 2.1 [3]:

```
cbench -c localhost -p 6633 -m 1000 -l 20 -s 50 -M 500 -t
```

Option	Definition
-c	controller address
-p	controller listening address
-m	duration
-l	number of loops
-s	number of switches
-M	number of hosts/MACs per switch
-t	throughout mode

Table 2.1: Cbench Options

2.15.3 Iperf

This is a command-line utility which tests the bandwidth between hosts. By the help of iperf, the performance of the controller would be measured. Iperf is a built-in feature in Mininet that works based on an iperf TCP server and another iperf TCP client on two virtual hosts. Once they are setup, they start blowing up packets between each other and iperf calculates the bandwidth between them [74].

2.15.4 Wireshark

Wireshark is an open-source software for analyzing packets and troubleshooting in the network. It is used on a wide range of Unix/Linux based operating systems. Wireshark knows about a variety of networking protocols and can interpret specific packets among them. It works perfectly live and captures filtered data and displays the results via its Graphical User Interface (GUI) [1]. Wireshark is going to be used in combination with Tcpdump further in this thesis.

2.15.5 Tcpdump

Tcpdump is another tool for analyzing packets which is command line free software. It is also used on Unix/Linux based systems as well as Windows systems. Tcpdump presents behavior of the network and network infrastructure by showing connectivity and activity which lead to further analysis [47].

Tcpdump tests would be done in combination with Wireshark tests to show the up/down links and how the flows travel between special nodes.

Chapter 3

Approach

This chapter gives a particular overview of the main test environment and settings as well as introducing different network graphs that are going to be used in tests. Each network graph is based on a Networkx command written in a python script which can be found in Appendix chapter. The main idea is to expand each graph topology to find a pattern which defines the relationship between the topology growth and bandwidth usage and consequently to compare different topologies (classic and random) in terms of their patterns.

3.1 Testbed design

3.1.1 Virtual environment and Mininet

In order to have the best performance, Mininet virtual environment is chosen in this thesis. It is installed as a virtual machine on Vmware Workstation. General specifications of the virtual environment are shown in Table 3.1 .

3.1.2 Networkx

Networkx as the main tool for construction and manipulation of complex topology networks can be downloaded via hg clone at Mercurial source code repository. Networkx is based on Python and a networkx module is recommended to begin with as below:

```
import networkx as NX
```

Case	Specification
Vmware version	8.0
Mininet package	mininet-vm-ubuntu11.10-052312
Virtual machine RAM	3 GB
Virtual machine HDD	8 GB
Virtual NIC	Default NAT
Number of processor(s)	One
Special Service	SSH auto login enabled
Switch Type	OpenVswitch

Table 3.1: Virtual Environment Specifications

Creating a graph using NX module is straight forward as below (this makes a ladder graph with corresponding options:

```
G = NX.ladder_graph(100, create_using=None)
```

Using Python scripting it is possible to change the graph to an Adjacency matrix as an example and save it to a text file with a comma delimiter (See Appendix A.1)

3.2 Network graphs design

Below the network graphs that are going to examine with Networkx are introduced with corresponding options. These graphs are divided into two categories of classic and random which are going to be compared further.

3.2.1 Classic graphs design

1. **Balanced Tree** - Function below gives a perfect balanced tree in Networkx:

   ```
   balanced_tree(r, h, create_using=None)
   ```

   Below items define each of the parameters for the function above:

   - **r :** (integer) is the width or branch of the tree
   - **h :** (integer) is the hight or depth of the tree
   - **create_using :** (optional) specifies type of graph (default is

networkx.Graph)

2. **Ladder** - Function below gives a ladder graph of pair nodes :

   ladder_graph(n, create_using=None)

   - **n :** (integer) is the length of the ladder
   - **create_using :** (optional) specifies type of graph (default is networkx.Graph)

3. **Star** - Function below gives a star graph with one central node connected to other nodes.

   star_graph(n, create_using=None)

   - **n :** (integer) is the number of outer nodes (total number of nodes n+1)
   - **create_using :** (optional) specifies type of graph (default is networkx.Graph)

3.2.2 Random graphs design

1. **Fast gnp** - Function below gives a random Erdos-Renyi binomial graph:

   fast_gnp_random_graph(n, p, seed=None, directed=False)

   - **n:** (integer) is the number of nodes
   - **p:** (float) is the probability for constructing an edge
   - **seed:** (integer) is an optional random number (defaults is none)
   - **directed:** (boolean) is optional and if it is true the graph is directed

2. **Powerlaw-Cluster** - Function below gives a random Powerlaw-cluster graph:

   powerlaw_cluster_graph(n, m, p, seed=None)

   - **n:** (integer) is the number of nodes
   - **m:** (integer) is the number of random edges for each new node

- **p:** (float) is the probability for adding a triangle after adding an edge

- **seed:** (integer) is an optional random number (defaults is none)

3. **Watts Strogatz** - Function below gives a Watts Strogatz small world graph:

   watts_strogatz_graph(n, k, p, seed=None)

   - **n:** (integer) is the number of nodes

   - **k:** (integer) is the number of closest neighbors in a ring shape topology

   - **p:** (float) is the probability for rewiring an edge

   - **seed:** (integer) is an optional random number (defaults is none)

4. **Random Regular** - Function below gives a random regular graph with degree of d:

   random_regular_graph(d, n, seed=None)[source]

   - **d:** (integer) is the degree of the nodes

   - **n:** (integer) is the number of the nodes where nxd is even

   - **seed:** (integer) is an optional random number (defaults is none)

5. **Barabasi Albert** - Function below gives a random graph with Barabasi Albert attachment model:

   barabasi_albert_graph(n, m, seed=None)

   - **n:** (integer) is the number of the nodes

   - **m:** (integer) is the number of edges as attachments to the existing nodes

   - **seed:** (integer) is an optional random number (defaults is none)

3.3 Mathematica functions and specifications

Wolfram Mathematica 8.0.0.0 is used in this thesis. Below explains the main mathematica commands and functions that have been used in combination with Networkx to work with network graphs (See Appendix A.3 for the whole commands):

- **Import** - to import a data file (adjacency matrix in this case)
- **AdjacencyGraph[a]** - to draw and make an object graph of the adjacency matrix
- **BetweennessCentrality[g]** - to extract node betweenness centrality list(based on number of shortest paths that pass through the nodes of the graph)
- **VertexDegree[g]** - to make a node degree list out of a graph object
- **DiagonalMatrix[VertexDegree[g]]** - to make a diagonal degree matrix
- **N[Eigenvalues[a]]** - to give numeric eigenvalues of matrix a
- **Normalize[e1]** - to normalize the sorted eigenvalues (with Sum equals to 1)
- **SmoothHistogram[ne1]** - to draw a well-shaped histogram of eigenvalues

3.4 Graph experiments methodology

Creating network topologies would be under Mininet and Networkx as the main testbeds and the data would be transferred to Mathematica for further calculations. The main features of graph tests are as below:

- Scaling up - up to 5 times from 10 to 160 main nodes (each time multiplied by 2)
- Repetition - 20 times for each random graph at each scaling (20x5)
- Eigenvalues and node degrees calculations

Further in Analysis and Discussion, how these features connect to network topology performance features and the possibility of giving a topology

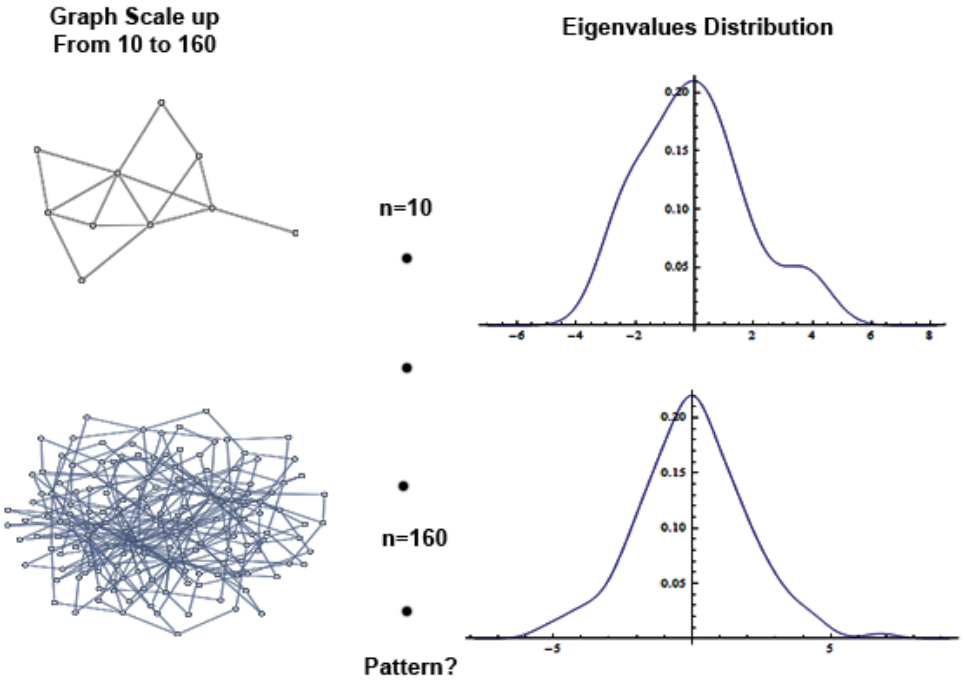

Figure 3.1: Graph pattern extraction methodology

growth pattern would be considered. The pattern extraction method follows the methods below:

- A possible pattern for each specific graph in growth
- A possible pattern for each graph type (classic and random) in growth
- A possible pattern for all graph types in growth

Figure 3.1 shows how the main methodology of graph investigation would be shaped.

3.5 Openflow Floodlight controller

Floodlight has been chosen as a business controller in this thesis with a variety of multi-purpose modules. As this controller is java-based, it needs default java tools as pre-requisites. In order to be compatible with Mininet topologies, some python tools are also needed. Another requirement is Apache ant that is a java library and is used to make Floodlight software in installation process. Below is a list of all required packages:

- build-essential
- default-jdk
- ant
- python-dev

Floodlight version 0.90 is used in this thesis which was the latest version available at Github. While simulating a network with Floodlight and Mininet, the controller has to be mentioned as remote with default Floodlight port which is 6633. The controller should be run beforehand with the command below in the Floodlight installed directory:

```
java -jar floodlight.jar
```

Below is a simple network running in Mininet which is pointing at Floodlight controller:

```
mn --controller remote --ip <floodlight ip> --port 6633
```

Topologies can also be given within the command above to be examined with the given controller.

Floodlight has a lot of modules that are loaded by default and make it heavy for tests. In order to optimize the tests, a minimized and optimal Floodlight is used with the loaded modules in floodlightdefault.properties file as below:

- net.floodlightcontroller.learningswitch.LearningSwitch
- net.floodlightcontroller.counter.NullCounterStore
- net.floodlightcontroller.perfmon.NullPktInProcessingTime

3.6 Openflow Pox controller

Pox has been chosen as the new generation of Nox to be considered as a scientific controller. As this controller is python-based, it requires Python 2.7 for the best performance. Pox beta version is accessible via Git repository and its default working port is 6634. Before referring to Pox at Mininet, it should be on by the minimal and optimized command as below:

```
./pox.py samples.pretty_log forwarding.l2_learning
```

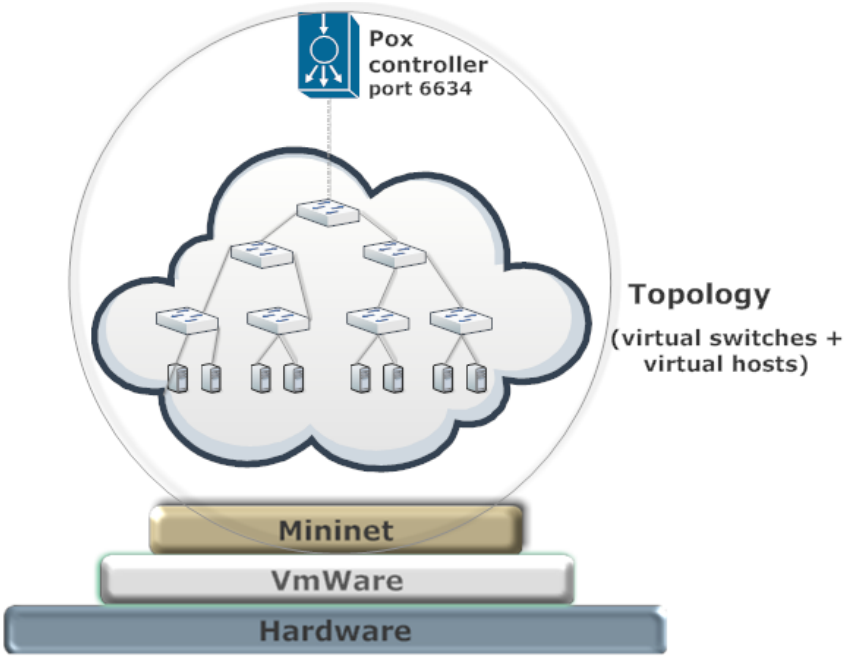

Figure 3.2: Conceptual Methodology

This brings up Pox with layer 2 learning module and logging enabled to perform optimal tests.

Figure 3.2 depicts the conceptual methodology of the thesis and its approach. The part within the circle is the Mininet operation range which contains the controller as well as the controller that is installed on Mininet virtual machine. The rest of beneath layers show the infrastructure of a virtual environment which is setup on top of the hardware layer to define an abstraction layer of network virtualization.

3.7 Openflow test tools specifications

To begin with the Openflow tests, first a comparison between two main powerful Openflow controllers would be done to chose one to go further with. The chosen controller would be the base controller to examine a variety of topologies (network graphs) with. In order to form such a test a combination of Cbench and Iperf tests would be done. Figure 3.3 shows the methodology that is used in Iperf approach to measure the performance of the controller by the help of the bandwidth between two hosts in an undirect manner. One of the hosts is considered as the Iperf server and the other as the client. Iperf is done by default settings but with the Cbench Table 3.2 shows the test specifications.

Case	Specification
Running mode	throughput
Controller Port	6633/6634
Faking 16 switches	2000 ms per test
MACs per switch	1000
Starting test delay	0 ms after features_reply
Ignoring	first 1 "warmup",last 0 "cooldown"
Connection delay	0ms per 1 switch(es)
Debugging info	off
Learning destination mac	before the test

Table 3.2: Cbench Specifications

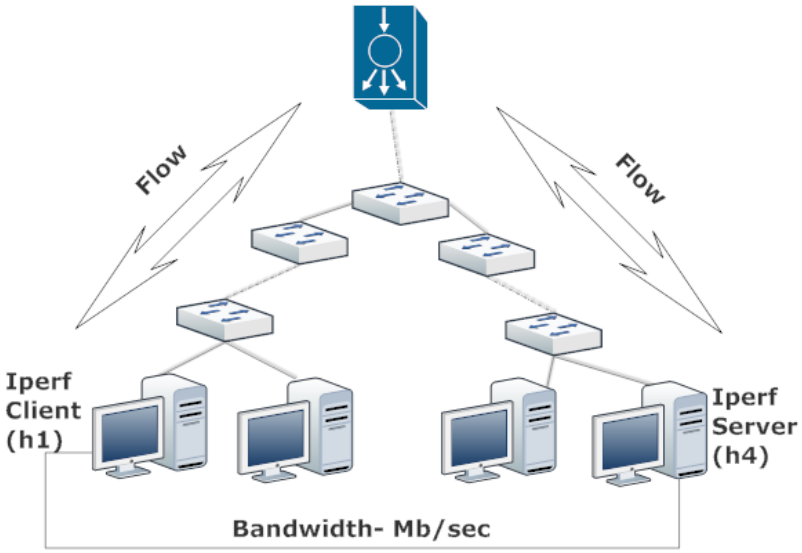

Figure 3.3: Iperf Methodology

Further in topology tests Wireshark and Tcpdump will be used to debug and perform some connectivity tests in case of failure in network and iperf will be used mainly to check the bandwidth between two given nodes which will examine the performance of the topology in an undirect manner as well.

Chapter 4

Results and Analysis

In this chapter a prototype architecture following by a summarized result of each network graph that has been created by Networkx and calculated by Mathematica would be given. Some important parts of the construction scripts and raw output would be considered as well. Since growing graphs lead to very big matrices, some of them have to be omitted from mentioning in this chapter. Instead, the mean distribution graphs would be given further.

Thus, to begin with the main mathematical equations that are used in the math scripts are introduced, following by Openflow controllers comparisons and corresponding topology tests with the chosen Openflow controller. The results of both theoretical and practical graph experiments will come further. More discussions and future work come in the next chapters.

4.1 Results and graphs architecture

In order to get the final results which are the benefits and drawbacks of the topologies and possible predictions in scaled up patterns, the processes in Figure 4.1 show the trend and the thorough architecture of the result prototype in this thesis.

Below, a sample Python code of a graph generation has been given. All other graph types are generated with slight changes in the graph generation part in the code (see Appendix A.1).

Listing 4.1: adja.py

```
1 #!/usr/bin/python
2
3 import networkx as NX # import networkx
```

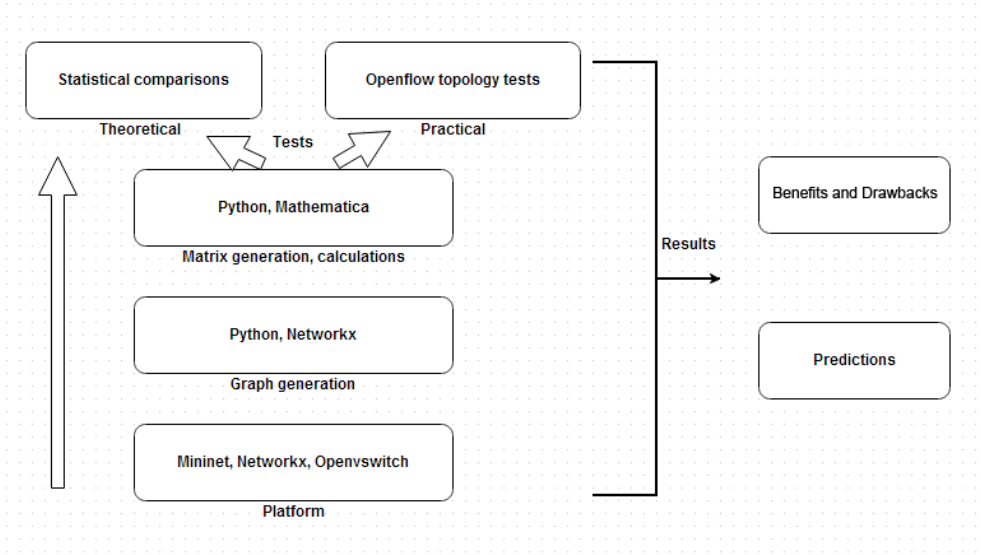

Figure 4.1: Results Architecture

```
4
5  # create a graph using a built-in graph generator from networkx
6
7  G = NX.barabasi_albert_graph(10, 2, seed=None)
8  type(G)
9
10 #<class 'networkx.classes.digraph.DiGraph'>
11 # express the graph as an Adjacency Matrix
12
13 AM = NX.to_numpy_matrix(G)
14
15 # use a built-in function from NumPy
16 # to save the Adjacency Matrix as a text file
17
18 import numpy as NP # import the library
19 NP.savetxt("s2/bar10.txt", AM, delimiter=',', newline="\n", fmt='%d')
```

4.2 Mathematical equations - adjacency histograms

In the Approach chapter, it was talked about the ways an adjacency matrix, node degree matrix and the corresponding laplacian matrix can extract from each other. The equations below are used basically on the results of Networkx graphs scripts to extract the matrices above practically. Later on, end results are done via script loops of histograms and distributions in combination with statistical equations.

$$adj = Import["bt2.txt","Data"] \tag{4.1}$$
$$hist = SmoothHistogram[adj] \tag{4.2}$$

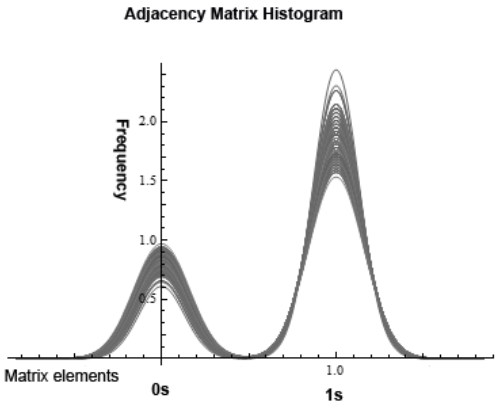

Figure 4.2: Matrix histogram sample

$$
\begin{aligned}
graph &= AdjacencyGraph[adj] && (4.3)\\
betweenness &= Sort[BetweennessCentrality[g]] && (4.4)\\
avgdegree &= N[Mean[VertexDegree[g]]] && (4.5)\\
degree &= DiagonalMatrix[VertexDegree[graph]] && (4.6)\\
laplace &= degree - adj && (4.7)\\
eigen1 &= N[Eigenvalues[adj]] && (4.8)\\
sort &= Sort[eigen1] && (4.9)\\
neigen1 &= Normalize[eigen1] && (4.10)\\
eigen2 &= N[Eigenvalues[laplac]] && (4.11)
\end{aligned}
$$

Matrix histogram is a Mathematica module that is used to give a visual view of large adjacency matrices. This considers matrix elements (0s and 1s) in a cumulative distribution function which smoothly shows the distribution of 0s and 1s in large matrices of 160x160 as an example here. This is taken as a way of summarizing and visualizing data as well as the ability to show the built in ability of graphs by the number of connections (1s) they have. Figure 4.2 is a sample of such histograms.

4.3 Classic network graphs results (prototype)

$$r = 4, h = 2, root = 1, nodes = 2 + (2^4) + 1 = 21 \qquad (4.12)$$

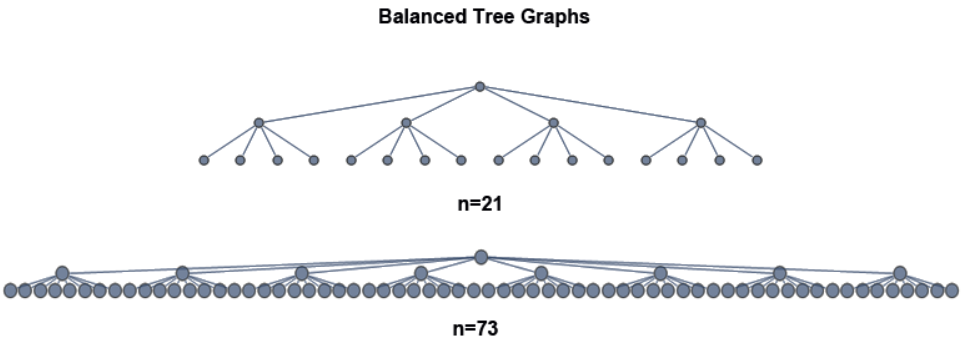

Figure 4.3: Balanced Tree Graph Scaling

4.3.1 Balanced Tree node calculation

This type of graph has two main variables; branching and height. In order to face with less complexity the variable height (h) is considered fixed with the value of 2 in all tests. The variable for branching (r) would go up from 2,4,8 to 12 to give the approximate required nodes. Below is an equation which shows how the number of nodes are calculated following with the corresponding command that is used in the Python script to generate such graph:

```
G = NX.balanced_tree(4, 2, create_using=None)
```

Table 4.1 shows balanced tree node scaling in this test.

Branch	Height	# of nodes
2	2	7
4	2	21
8	2	73
12	2	157

Table 4.1: Balanced Tree Node Scaling

4.3.2 Balanced Tree graph and matrix generation

Figure 4.3 shows a sample of graph scaling for 21 and 73 nodes that are mentioned above in terms of network graphs. Each time the number of branching has become twice(except the last experiment as it did not have to go up 160 nodes to stay in match with other tests) the number of nodes follow the Table 4.1.

According to Appendix A.3 adjacency graphs, degree and laplacian

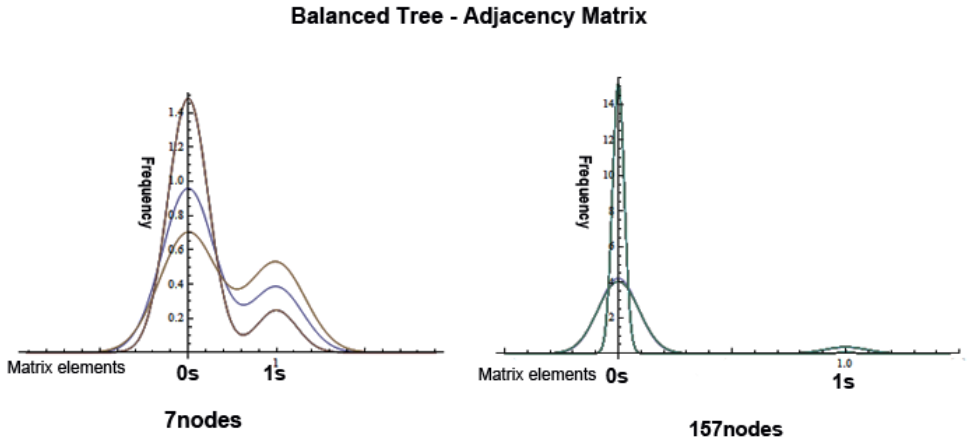

Figure 4.4: Balanced Tree Adjacency Matrix histograms

matrices,Eigenvalues and laplacian Eigenvalues are calculated for each set of tests. Matrices in this type of graph start with 7x7 dimensions and end with 153x153(See Appendix B.1 for an output of 21x21 matrix of a balanced tree). Figure 4.4 shows a sample of scaled up histograms for each of nxn matrices (n=2,12). These histograms give a better overview of the number of 0s or 1s being distributed in each matrix (row by row). For complete graphs and matrix histograms see Appendix B.2.

Nodes	Max eigen	Min Lap eigen	Avg Degree	Min Degree	Betweenness
7	2	0.39	1.71	1	9
21	2.82	0.17	1.90	1	150
73	4	0.10	1.97	1	2268
157	4.89	0.07	1.98	1	11154

Table 4.2: Balanced Tree Eigenvalues and Node Degrees

Maximum eigenvalue of an adjacency matrix shows epidemic information spreading for the corresponding node and the whole graph generally while betweenness centrality gives specific measurements about importance and load of a node. When it comes to laplacian Eigenvalues, the second minimum laplacian Eigenvalues are the most important values that can imply the algebraic connectivity. Average vertex(node) degree of a graph is also interesting to know about the robustness of the system. Table 4.2 gives different Eigenvalues that are extracted for each set of tests and the node degrees of the whole graph at each test.

4.3.3 Ladder graphs and matrices

A ladder graph with n as the number of nodes in the command returns a pair connected graph of $2n$. This means in order to have the node scaling of

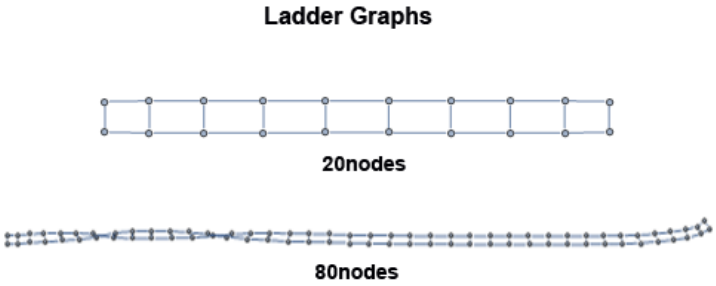

Figure 4.5: Ladder Graph Scaling

10, 20, 40, 80 and 160 we need an n of 5, 10, 20, 40 and 80. The equation and the command below is a Networkx ladder graph generator of 2x80 (160) nodes which has been used in Python script of graph generation:

$$n = 80, O(2 * n) = 2 * 80 = 160 \tag{4.13}$$

```
G = NX.ladder_graph(80, create_using=None)
```

Ladder graphs grow in length and pair by pair which makes the shape of the graphs and format of the matrices similar (See Figure 4.5 and 4.6), but the increase in length might change the study of distance in the network. As it is shown in Figure 4.6 the distribution of 0s in Ladder graph matrices are importantly more than the distribution of 1s.

Table 4.3 shows the results for different Eigenvalues ,node degrees and betweennesses that are calculated from the ladder graph adjacency matrix. For more Ladder sample results see Appendix B.5.

Nodes	Max eigen	Min Lap eigen	Avg Degree	Min Degree	Betweenness
5	2.73	0.381	2.6	2	10.7
10	2.91	0.097	2.8	2	46.1
20	2.97	0.024	2.8	2	193
40	2.99	0.006	2.95	2	786.9
80	3	0.001	2.95	2	3174.6

Table 4.3: Ladder Eigenvalues and Node Degrees

4.3.4 Star graphs and matrices

A star graph can be considered as one branch of a tree graph. Thus, the structure is quite simple and the growth follows O(n+1). But since it does

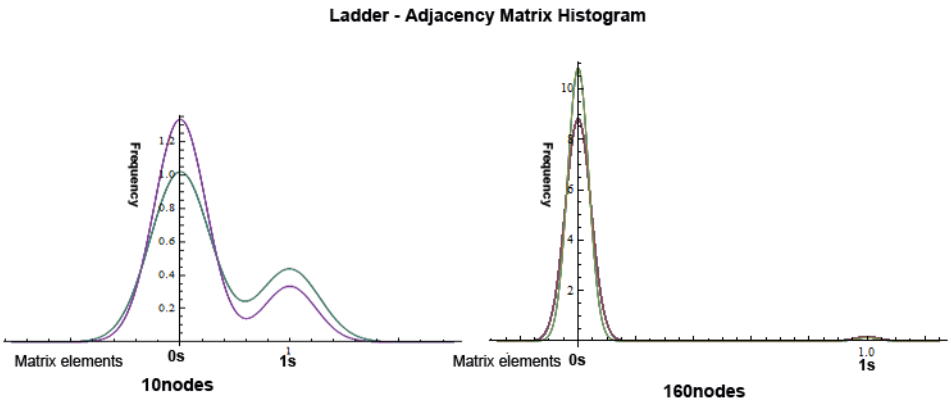

Figure 4.6: Ladder Adjacency Matrix Histograms

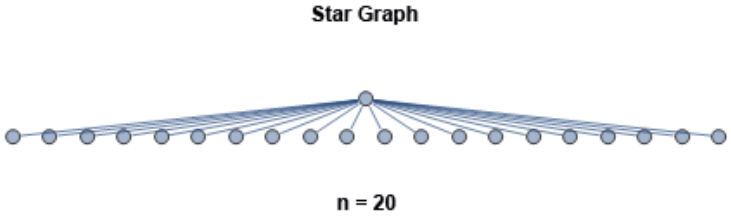

Figure 4.7: Star Graph Sample

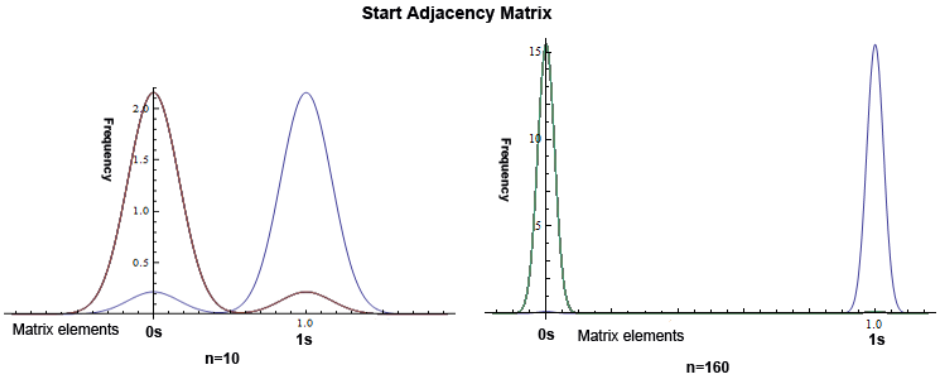

Figure 4.8: Star Adjacency Matrix Histograms

not have the concept of height, the growth goes through the width and this has its own circumstances to performance and bandwidth. Figure 4.7 depicts a sample of 20 nodes star graph.

The command below is a Networkx star graph generator of 160 nodes which has been used in Python script of graph generation:

```
G = NX.star_graph(160, create_using=None)
```

The adjacency matrix distribution in star graph shows a bit of a difference as it consist of 1 only in conjunction with the root node and all other connections are presented with 0s. That is the reason why the matrix histogram shows an M shape with thinner end at the 1, which represents the only connection to the root and all others are 0s. Figure 4.8 shows the distribution of 0s and 1s for 10 and 160 number of nodes.

The same results come up with the second minimum laplacian Eigenvalues and minimum node degree for the same reason of having the root architecture and no depth content. The amount of betweenness centrality is very hight in this type of graph. Table 4.4 gives a better overview of this model. For more Star sample results see Appendix B.8.

Nodes	Max eigen	Min Lap eigen	Avg Degree	Min Degree	Betweenness
10	3.16	1	1.81	1	45
20	4.47	1	1.90	1	190
40	6.32	1	1.95	1	780
80	8.94	1	1.97	1	3160
160	12.64	1	1.98	1	12720

Table 4.4: Star Eigenvalues and Node Degrees

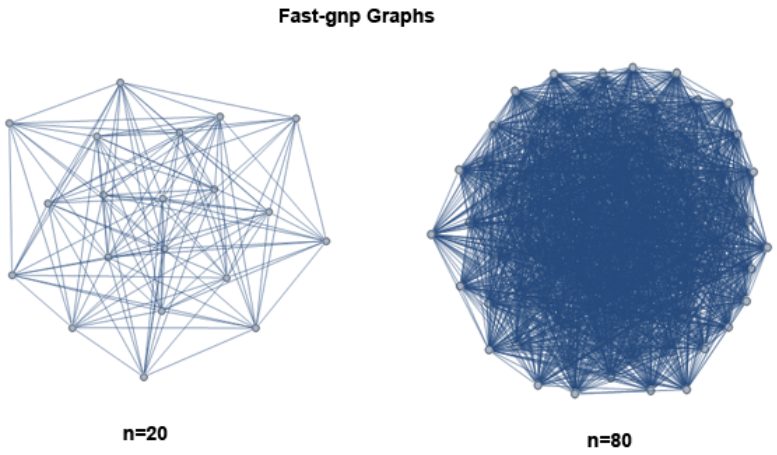

Figure 4.9: Fast-GNP Graphs

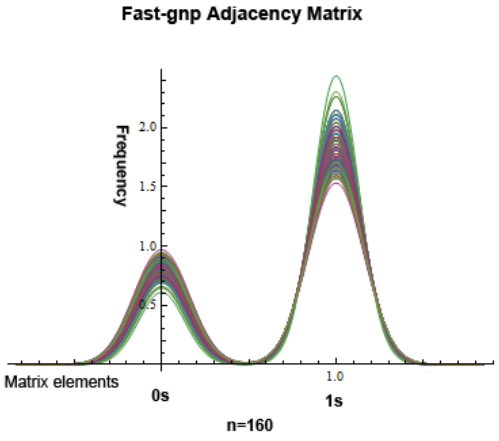

Figure 4.10: Fast-GNP Adjacency Matrix Histogram

4.4 Random network graphs results (prototype)

4.4.1 Fast-GNP random graphs and matrices

Fast GNP is a closely related Erdos-Renyi graph with some improvements in the speed of algorithm. The first difference that can be seen in such random graphs is the probability option. The higher the probability is, the more dense and complicated the graph is. Command below is a Networkx command which defines a random fast-gnp graph with probability of 0.7.

```
G = NX.fast_gnp_random_graph(10, 0.7, seed=None, directed=False)
```

The results of graph shapes are shown in Figure 4.9 which depicts how dense the graph would be when it scales up from 20 to 80 nodes. The high density of the graph connections show that there must be a visible change in the distribution of 1s in the adjacency matrix of the the graph. As it is shown in Figure 4.10 the pattern of 0s and 1s distributions in the adjacency matrix of a random fast-gnp has changed in a way that the frequency of 1s has increased. This pattern was slightly the same within all adjacency matrices while scaling up the graph. The increase in the distribution of 1s in the adjacency matrix is the proportion of the probability that has been taken for the graph generation.

Studying Eigenvalues, both maximum eigenvalue and second smallest laplacian eigenvalues show noticeable growth and node degrees specially minimum node degree has more variety while showing increases at the same time. Table 4.5 studies the values above as well as betweenness centrality in more detail. For more Fast-gnp sample results see Appendix B.4.

Nodes	Max eigen	Min Lap eigen	Avg Degree	Min Degree	Betweenness
10	6.80	4.29	6.06	5	6.3
20	13.43	8.65	13.2	10	17
40	26.32	17.70	25.95	19	22.1
80	56.07	44.27	55.85	46	54.1
160	111.24	94.72	110.95	98	76.2

Table 4.5: Fast-gnp Eigenvalues and Node Degrees

4.4.2 Powerlaw Cluster random graphs and matrices

Powerlaw Cluster graph algorithm tries to make triangle out of added edges. The triangle shapes are shown in Figure 4.11 as examples of powerlaw graphs. Command below shows how to make a powerlaw graph with 160 nodes and 4 random edges for each of the new nodes. Probability of 0.7 is for adding a triangle after adding a new edge.

```
G = NX.powerlaw_cluster_graph(160, 4, 0.7, seed=None)
```

Nodes	Max eigen	Min Lap eigen	Avg Degree	Min Degree	Betweenness
10	5.18	1.52	4.6	2	15
20	7.20	2.17	6.2	4	41.6
40	9.08	2	7.1	3	131.3
80	10.71	1.72	7.57	2	1039.8
160	12.93	1.57	7.72	3	2569.6

Table 4.6: Powerlaw Cluster Eigenvalues and Node Degrees

Powerlaw as a network graph shows a problem strengthening itself as the

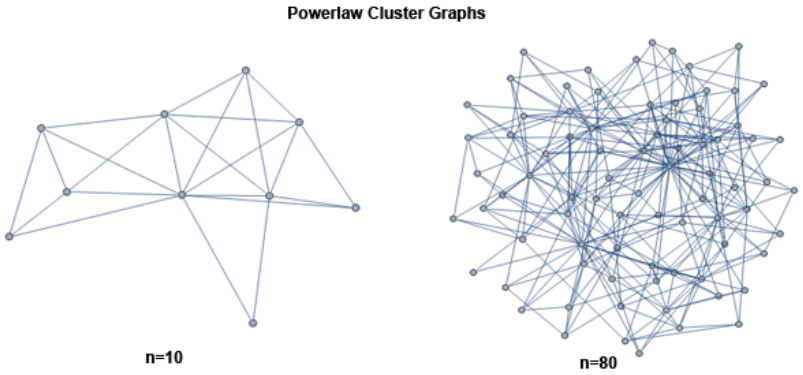

Figure 4.11: Powerlaw Cluster Graphs

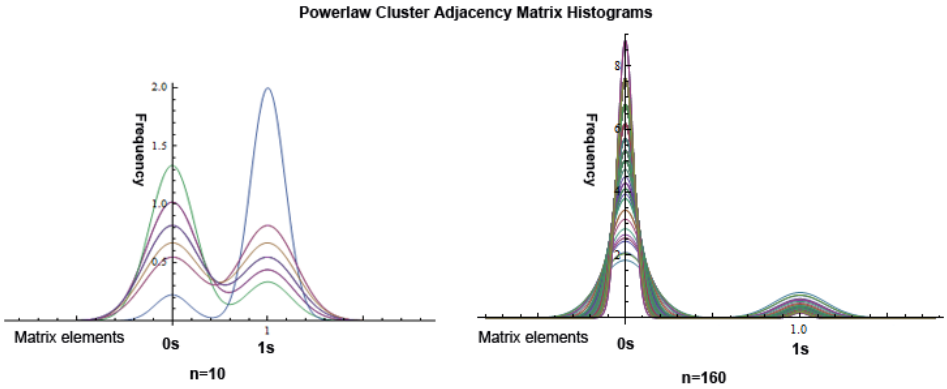

Figure 4.12: Powerlaw Cluster Adjacency Matrix Histograms

triangles fraction decreases by increasing the size of the network. This can be seen in changing the pattern of adjacency matrices while scaling up. Figure 4.12 shows this change in more detail where the ratio of 1s in the matrices drops in scaling. Probing the eigenvalues and node degrees, what is interesting is that there is no big change which is comparable to the scaling in the number of any of the variables. Especially node degrees do not show much growth unlike the betweenness which is rather big in this type of graph. Table 4.6 gives a detailed overview of the discussed variables. For more Powerlaw sample results see Appendix B.6.

4.4.3 Watts-Strogatz random graphs and matrices

To begin with, Watts-Strogatz algorithm makes a ring graph over n number of nodes. Later, each node will be connected to a number of closest neighbors which is defined by the variable k in the command below.

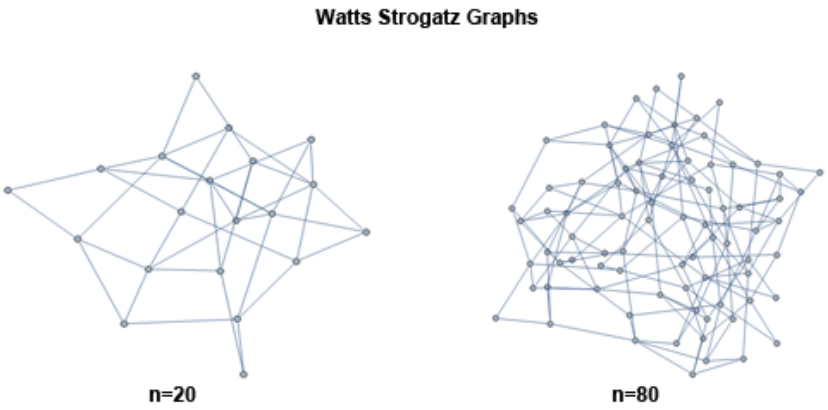

Figure 4.13: Watts-Strogatz Graphs

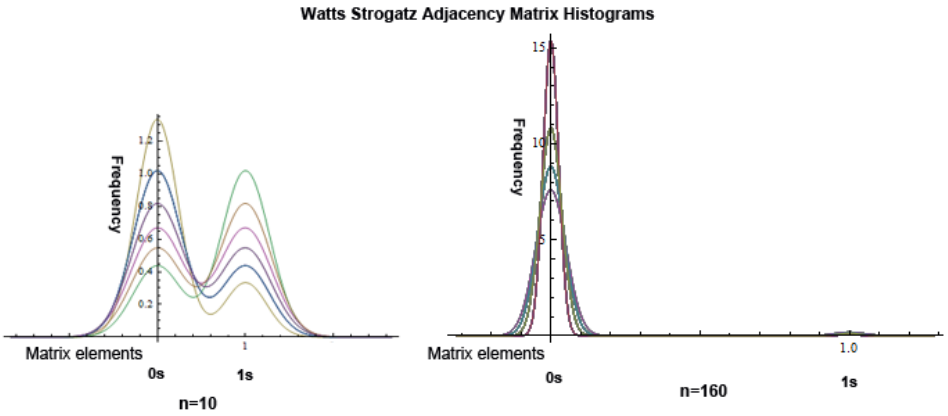

Figure 4.14: Watts-Strogatz Adjacency Matrix Histograms

Finally, some shortcuts will be created by adding up some edges. Edge making would be done by the given probability as below :

```
G = NX.watts_strogatz_graph(80, 4, 0.7, seed=None)
```

The command above shows how to make a watts-strogatz graph with 80 nodes connecting to 4 closest neighbors with edge creating probability of 0.7.

Figure 4.13 shows two sample graphs for n=20 and n=80. This type of graph with mentioned number of neighbors probability did not make a very complicated graph in terms of number of connections. Therefore the distribution of 0s in the adjacency matrices are bigger and this can be seen in Figure 4.14 which examines two histograms of two adjacency matrices for n=10 and n=160 respectively. What differentiates Watts-Strogatz graphs

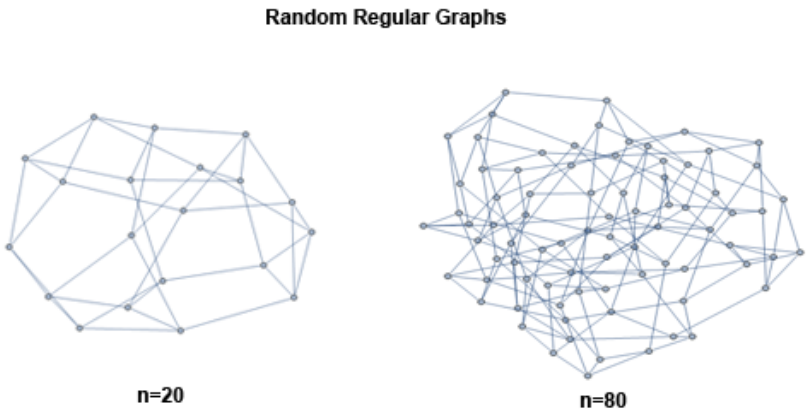

Figure 4.15: Random Regular Graphs

from the previous ones is that the average node degree and the min node degree variables stay constant while scaling up the network. This can be seen slightly in maximum eigenvalues as well unlike the betweenness which is rather big and grows well in this type of graph. Table 4.7 shows the results thoroughly. For more Watts sample results see Appendix B.9.

Nodes	Max eigen	Min Lap eigen	Avg Degree	Min Degree	Betweenness
10	4.43	1.47	4	2	11.8
20	4.45	1.08	4	2	25.9
40	4.65	0.76	4	2	108.6
80	4.43	0.59	4	2	257.9
160	4.57	0.56	4	2	1036.8

Table 4.7: Watts-Strogatz Eigenvalues and Node Degrees

4.4.4 Random Regular graphs and matrices

The first option which differentiates a random regular graph is degree option. In order to make the algorithm work properly the multiplication of degree and the number of nodes should be an even number. Below show a Networkx command that creates a 160 nodes random regular graph with degree of 4.

```
G = NX.random_regular_graph(4, 160, seed=None)
```

The Algorithm keeps this degree all over the graph regularly. Figure 4.15 shows sample graphs of random regular with 4 degrees for n=20 and n=80.

Adjacency matrices are very regular in this model. Thus the matrices'

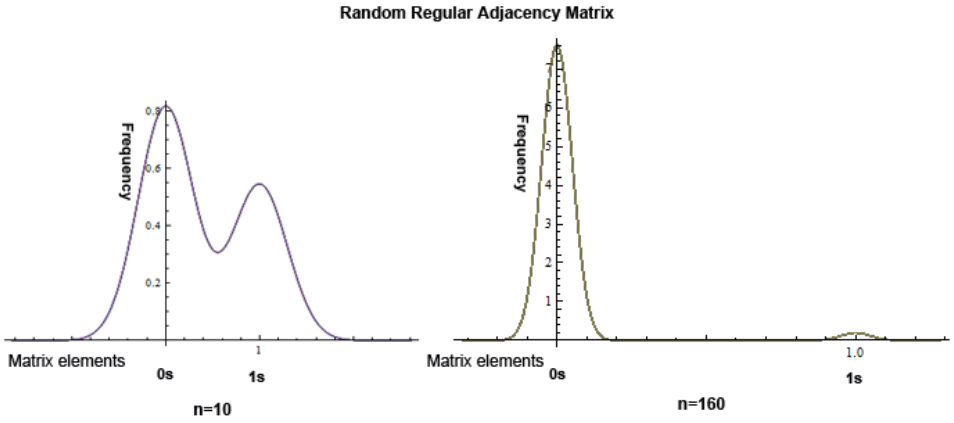

Figure 4.16: Random Regular Adjacency Matrix Histograms

histograms do not show variations in number of curves. Since the random connections are made to the close neighbors in this type of graph, the distribution of 0s in the matrices grow while scaling up. Figure 4.16 shows this fact by depicting adjacency matrix histograms of 10 and 160 nodes respectively.

The same regularity can be seen in eigenvalues and node degrees. Three columns out of 5 columns of Table 4.8 represent the same value. For more Regular sample results see Appendix B.7.

Nodes	Max eigen	Min Lap eigen	Avg Degree	Min Degree	Betweenness
10	4	2.13	4	4	3
20	4	1.10	4	4	17.1
40	4	0.91	4	4	49.2
80	4	0.66	4	4	143.7
160	4	0.57	4	4	320.6

Table 4.8: Random Regular Eigenvalues and Node Degrees

4.4.5 Barabasi-Albert graphs and matrices

A differentiation option about Barabasi-Albert graph is the number of edges for attaching the new node to previous nodes. Below is the Networkx command to make a barabasi graph of 160 nodes with 4 edges attaching to new nodes.

```
G = NX.barabasi_albert_graph(160, 4, seed=None)
```

Barabasi tries to create the specified number of edges, but there is no

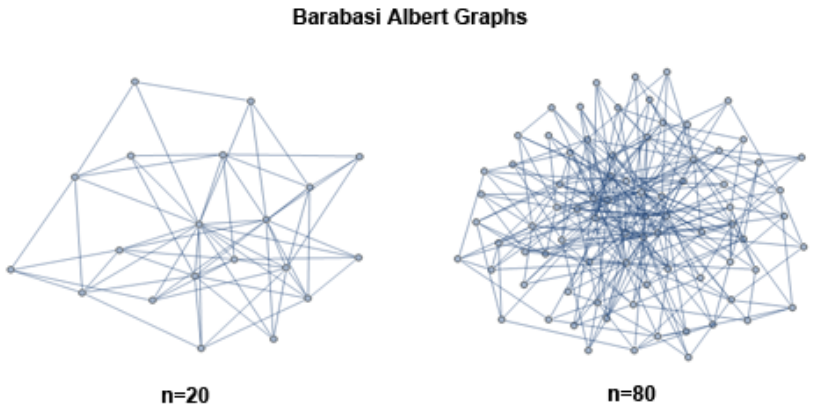

Figure 4.17: Barabasi Albert Graphs

absolute guarantee. As seen in Figure 4.17 for n=20 some nodes do not have 4 edges attached to them.

Nodes	Max eigen	Min Lap eigen	Avg Degree	Min Degree	Betweenness
10	5.26	2.34	4.8	3	10.7
20	7.49	2.21	6.4	3	49.9
40	9.25	2.06	7.2	4	165.3
80	10.69	2.17	7.6	4	476.2
160	12.79	2.15	7.8	4	1458.7

Table 4.9: Barabasi Albert Eigenvalues and Node Degrees

The Adjacency matrices for this model do not show major differences in scaling. Therefore, Figure 4.18 depicts a histogram of adjacency matrix for a 160x160 matrix.

The results in terms of eigenvalues and node degrees show reasonable values even though the number of edges were not always even 4 per node. Minimum node degree appear not to be less than 3 for each node while average node degree is always above 4. Betweenness centrality looks reasonable as well. Table 4.9 gives the above results in detail. For more Barabasi sample results see Appendix B.3.

4.5 Openflow Controller comparison results

In order to perform practical tests of network graphs on Openflow architecture a comparison between two powerful Openflow controllers has been taken to take one controller to go further with. Since both controllers

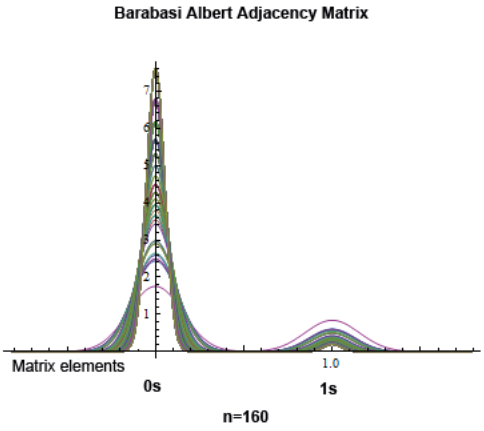

Figure 4.18: Barabasi Albert Adjacency Matrix Histogram

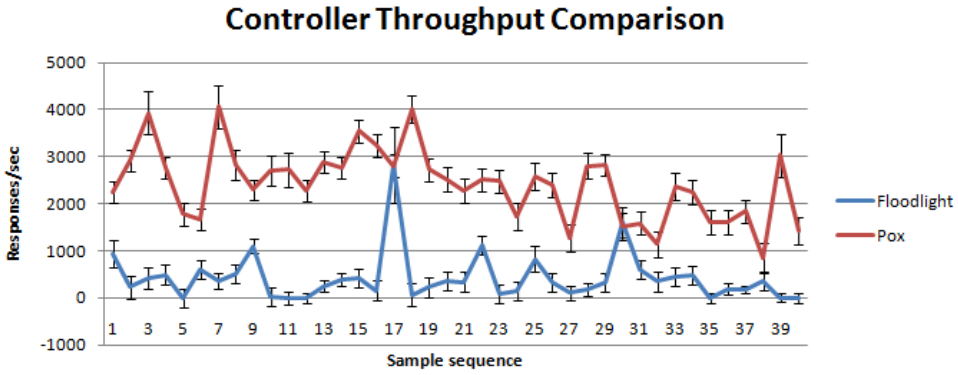

Figure 4.19: Pox versus Floodlight - Throughput

are modules based and are capable of loading many modules, the test has been taken with the minimum possible modules to let the controllers as light as possible and take the highest performance out of them. Test conditions and modules are described in Approach chapter. Below is the Cbench command which examines the controllers throughput.

```
cbench -c localhost -p 6633 -m 1000 -l 20 -s 50 -M 500 -t
```

The test has been done with 40 times repetitions and the results are the means of all the repetitions. Figure 4.19 shows the results. Error bars are twice the standard deviation which show slightly more fluctuation in Pox controller. However, Pox has better throughput in overall in comparison with Floodlight. Considering some zeros as throughput in the results of Floodlight which means the controller did not respond at the certain period time, the fluctuations in Pox can be ignored. Thus, for this type of test with random network topologies the lighter controller with higher throughput is picked to go further with in the next experiments.

4.6 Mininet topology scripts (prototype)

In order to test the exact graphs with the same matrices, python implementation scripts have been used. For both classic and random graphs, the exact topology has to be made in python node by node from the adjacency matrix.

Below is an implementation sample for balanced tree topology.

Listing 4.2: btree.py

```
#!/usr/bin/python

from mininet.topo import Topo, Node

class MyTopo( Topo ):
    "Simple topology example."

    def __init__( self, enable_all = True ):
        "Create custom topo."

        # Add default members to class.
        super( MyTopo, self ).__init__()

        # Set Node IDs for hosts and switches
        s1=1, s2=2, s3=3, s4=4, s5=5, s6=6, s7=7
        h1=11, h2=22, h3=33, h4=44, h5=55, h6=66, h7=77
        # Add nodes
        self.add_node( s1, Node( is_switch=True ) )
       #Continue the same way till s7
        self.add_node( s7, Node( is_switch=True ) )

        self.add_node( h1, Node( is_switch=False ) )
       #Continue the same way till h7
       self.add_node( h7, Node( is_switch=False ) )

        # Add edges based on Adjacency matrix
        self.add_edge( h1, s1 ), self.add_edge( h2, s2 )
        self.add_edge( h3, s3 ), self.add_edge( h4, s4 )
        self.add_edge( h5, s5 ), self.add_edge( h6, s6 )
        self.add_edge( h7, s7 )

        self.add_edge( s1, s2 ), self.add_edge( s1, s3 )
        self.add_edge( s2, s4 ), self.add_edge( s2, s5 )
        self.add_edge( s3, s6 ), self.add_edge( s3, s7 )

        # Consider all switches and hosts 'on'
        self.enable_all()
topos = { 'mytopo': ( lambda: MyTopo() ) }
```

For a complete version of matrix based topology scripts see Appendix A.2.

4.7 Considerable Circumstances for Analysis

The results that are taken till now are going to be analyzed in addition with some other extracted results. The fact is some special conditions might have affected the results and the corresponding analysis. Thus, it is wise to be aware of them before beginning any analysis. Below are some of the most important circumstances:

- Probability distribution

  This is one of the most important characteristics of a random graph. Tuning and picking one as a general probability distribution was not easy. This number can be between 0 and 1 as it is probability but choosing it below 0.5, it might end up with disconnected graph. This can be 9 out of 10 nodes connected and only one node disconnected or just 2 multiple nodes disconnected graphs and that happens purely random.

  In order to avoid disconnected graphs as much as possible, the probability distribution of 0.7 has been taken generally. Thus, all random graphs are well connected and look symmetric.

  This leads to always have the second minimum laplacian eigenvalue greater than zero which represents a well connected graph. This also leads to average node degree of not smaller than 4. Therefore, for random regular and barabasi graphs which picking a probability distribution was not possible due to Networkx command format, the average node degree of 4 has been chosen.

- **Openflow Testbed Problem**

  Openflow has been taken as the main platform for the graph experiments. The fact is, with such high probability distribution (0.7), there would be loops in the networks. Thus, there must be mechanisms to avoid the loops within the testbed.

  Since the platform is a simulated testbed (Minimet) with added controller (Pox), any results and circumstances might be due to this configuration.

  Pox uses a module called openflow.spanning_tree which takes place after knowing about the topology of the network and provides a Spanning Tree to have a loop free and more robust network. Unfortunately, using this module did not give much of a help to complex loops of random graphs.

  What actually happened during the experiments was more like a

vague try and error which kept closing more and more ports at each time trying. The results at a network with about 2 to 3 ports to be closed to avoid the loops was closing more than 80 percent of ports.

The experiments with ping showed that the ports that were open on one side were not open any more on the other side and that could be when for instance h1 could ping h3 at the moment, at another moment h3 could not ping h1 because each time the packets would be sent to the controller to decide and for some reasons the loop avoidance module of the controller did not work properly and simply closed most of the ports.

That meant disaster to do the rest of the thesis with this platform. Other controller got tested as well and the same results were taken more or less. Since no real network devices were in hand, the strategy was to use the same platform.

Networkx and Python programming have been the key to this problem. Thus, a spanning tree minimal code has been added to the topology creation and minimum loops have been avoided by the help of Networkx and its compatibility with Python and Mininet. Below is the minimum loop avoidance code which runs an internal spanning tree protocol for a fast-gnp graph:

Listing 4.3: stp.py

```
1  #!/usr/bin/python
2
3  G = NX.fast_gnp_random_graph(10, 0.7, seed=None, directed=False
4  type(G)
5
6  AM = NX.to_numpy_matrix(G)
7
8  D=NX.DiGraph(G)
9
10 D.remove_edges_from(G.selfloop_edges())
```

Therefore, Network graphs are still tested on Openflow platform and measurements are taken from openflow controller but the controller was not capable of avoiding complex loops which appear in random graphs.

This could be a drawback of a simulated platform or having a loop avoidance mechanism being centralised inside a controller. Finding out the reasons can be a subject to another research.

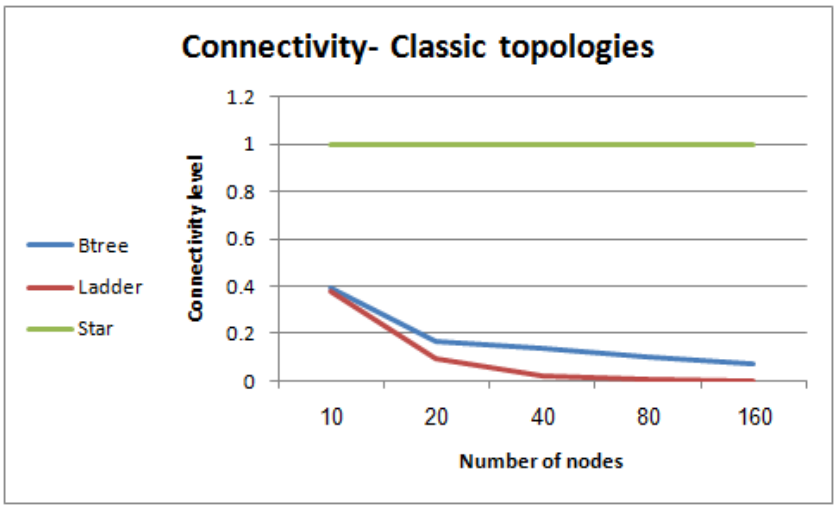

Figure 4.20: Algebraic Connectivity - Classic graphs

4.8 Algebraic Connectivity - Term Analysis

The term algebraic connectivity exists in Graph theory and refers to second smallest laplacian eigenvalue of a graph. If this value is greater than zero, the graph is considered as connected and the amount of the value show how well the graph is connected. This has been used to compare different graphs and corresponding topologies in theory and in practice in this section.

4.8.1 A.Connectivity -Theoretical analysis of classic graphs

Considering second minimum Laplacian Eigenvalue as Algebraic Connectivity, the results in Figure 4.20 are taken from classic graphs scaling up from 10 to 160 nodes.

The term algebraic connectivity shows how well connected a graph is. As it is shown in Figure 4.20 the star graph is the most well connected classic graph. The reason can be that all the nodes are connected to the root node and are only one node apart from each other. Star is the most flat graph in this category. Table 4.10 shows the prioritization among the classic graphs in terms of connectivity.

Prioritization	Graph
1	Star
2	Btree
3	Ladder

Table 4.10: Classic graphs Connectivity Prioritization

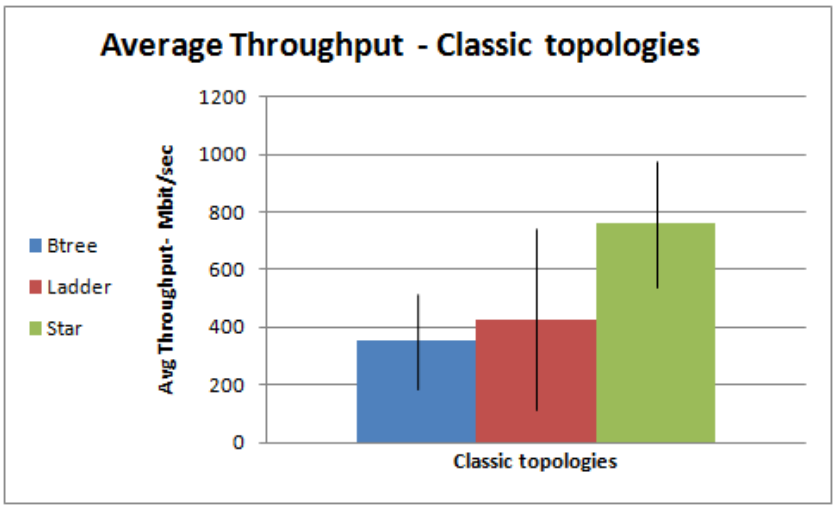

Figure 4.21: Average Throughput - Classic Graphs

4.8.2 A.Connectivity -Practical analysis of classic topologies

Considering the same feature, a practical experiment has been taken on Openflow platform to measure throughput of the same classic graphs. Figure 4.21 shows the results of the experiments. The more a graph is connected the better it is performed. This can be proved by the practical experiment as well.

The tests have been done 30 times and the results are showing the average throughput in milliseconds. Error bars are twice the standard deviation and the tests have been taken under a stressed system by transferring large packets between nodes. The Ladder graph shows more fluctuation in performance and this could be because of the long structure of the ladder that might cause some problems in practice.

Table 4.11 shows the prioritization among the classic graphs in terms of connectivity in practice which is considered as throughput.

Prioritization	Graph
1	Star
2	Btree
3	Ladder

Table 4.11: Classic graphs Throughput Prioritization

As it is shown, the results show the exact match in prioritization.

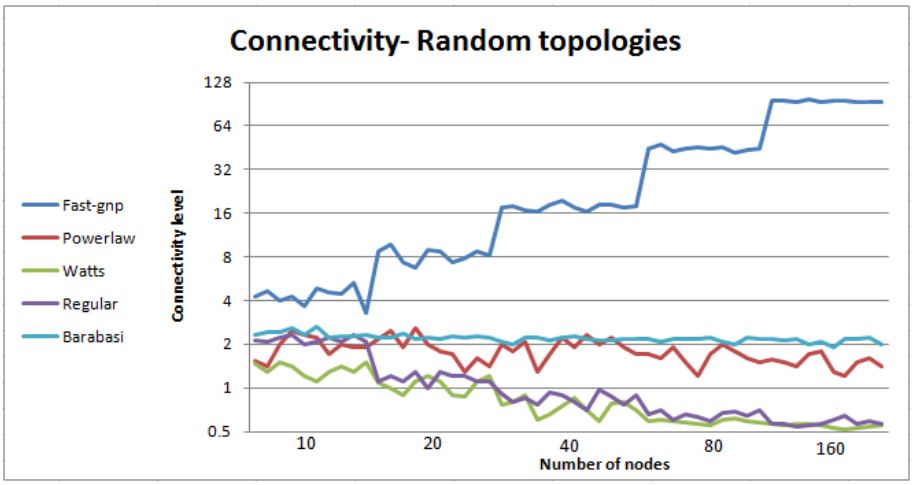

Figure 4.22: Algebraic Connectivity - Random Graphs

4.8.3 A. Connectivity -Theoretical analysis of random graphs

Comparing second minimum laplacian eigenvalue of the random graphs, Figure 4.22 shows the theoretical results. This shows the theoretical trend of random graphs in terms of connectivity while scaling up. Studying the practical trend of such graphs might end up with similar or different results.

The Theoretical results show the prioritization as is in Table 4.12.

Prioritization	Graph
1	Fast-gnp
2	Barabasi
3	Powerlaw
4	Regular
5	Watts

Table 4.12: Random graphs Throughput Prioritization

4.8.4 A. Connectivity -Practical analysis of random topologies

Applying the same performance test to random topologies while making them by corresponding python scripts, the results in Figure 4.23 are taken.

Prioritizing the results, it ends up with Table 4.13 which ranks the random graphs in practice.

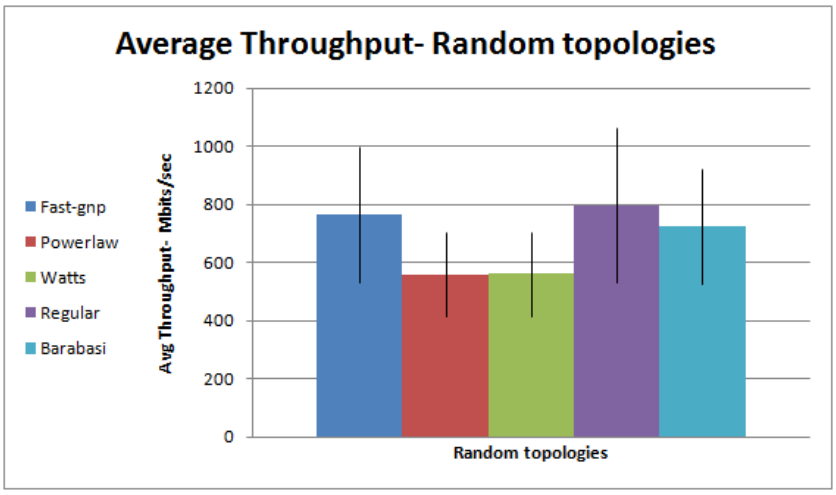

Figure 4.23: Average Throughput - Random Graphs

Prioritization	Graph
1	Regular
2	Fast-gnp
3	Barabasi
4	Watts
5	Powerlaw

Table 4.13: Random graphs Throughput Prioritization

Comparing Table 4.12 and Table 4.13, a mismatch between theoretical and practical results of random graphs in terms of algebraic connectivity can be recognised. Since the practical results have been taken in acceptable number of repetitions, they are most likely trustable. What might have caused this mismatch could be one or both of the following :

- Random graphs behave different than classic graphs in terms of connectivity
- Algebraic connectivity is not a suitable feature to measure the performance of a random topology

4.9 Relative Robustness - Term Analysis

The term relative robustness with the following definition has not been used widely by now (as far as the author of the thesis know).

The term *average node degree* comes with the theory of graphs and becomes especially important about random graphs. When the network grows, it is

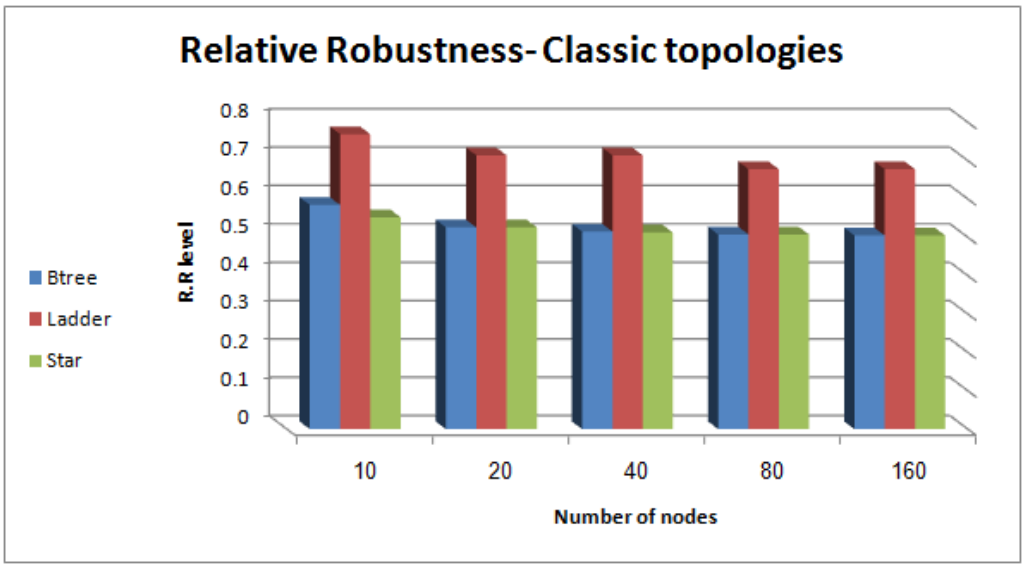

Figure 4.24: Relative Robustness - Classic graphs

essentially important to keep the average node degree to a certain level so that all the network behaves harmonically. This is discussed as network's robustness and as a product of probability distribution (edges) is one of the most characteristics of the random graphs.

On the other hand, another term which is called *minimum node degree* can affect the performance of the network as well. Suppose two network topologies with the same average node degrees but different minimum node degrees. Minimum node degrees without being too often (that affect the average node degree) can still affect the robustness especially in equal average situations. Experiments by the author of the thesis show that the topologies with the higher minimum node degree would perform better than the others.

These results ended up with an idea of defining a new term which is called ***Relative Robustness (R.R)*** with the following formula where the greater the R.R is, the better the topology perform:

$$R.R = MinNodeDegree / AvgNodeDegree \tag{4.14}$$

4.9.1 R. Robustness -Theoretical analysis of classic graphs

Considering the term above and applying that to the theoretical results, Figure 4.24 shows the results.

As it is shown in the results the prioritization among the classic graphs in

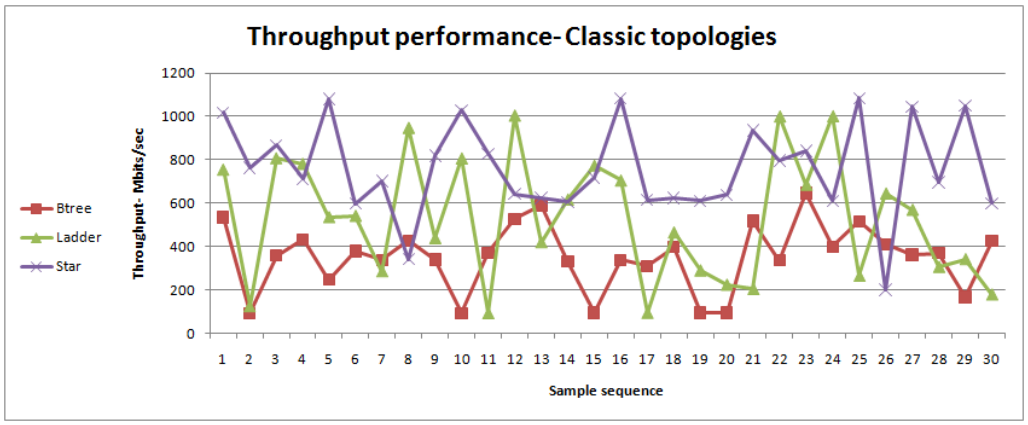

Figure 4.25: iperf Throughput - Classic Topologies

terms of relative robustness in theory can be seen in Table 4.14.

Prioritization	Graph
1	Ladder
2	Btree
3	Star

Table 4.14: Classic graphs Theoretical Relative Robustness Prioritization

4.9.2 R. Robustness -Practical analysis of classic topologies

As it was discussed before, classic topologies have been tested with iperf in terms of performance. The results are also shown in Figure 4.25 which gives a prioritization of the Star, Ladder and Balanced tree topologies.

This implies another mismatch between the theoretical predictions and practical results and it could mean one or both of the followings:

- Classic graphs behave different in terms of relative robustness
- Relative Robustness is not a suitable feature to measure the performance of a random topology

4.9.3 R. Robustness -Theoretical analysis of random graphs

The idea of relative robustness came from random graphs in medium to large networks. How it works in scaling up networks is shown in Figure 4.26. It is also showing the theoretical prioritization of random graphs which are summarized in Table 4.15.

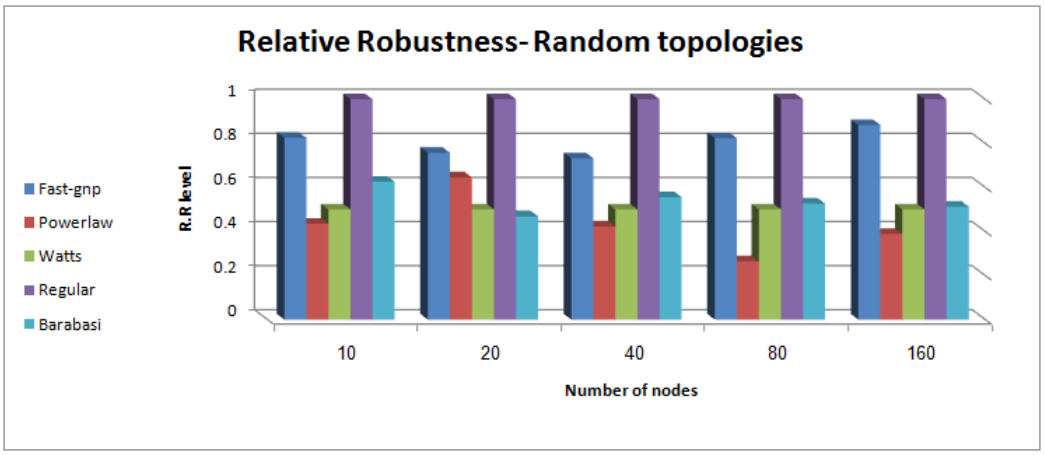

Figure 4.26: Theoretical Relative Robustness of Random Graphs

Prioritization	Graph
1	Regular
2	Fast-gnp
3	Barabasi
4	Watts
5	Powerlaw

Table 4.15: Random graphs Relative Robustness Prioritization

4.9.4 R. Robustness -Practical analysis of random topologies

In order to experience the performance of random graphs in practice an iperf throughput test with 30 repetitions has been done. Figure 4.23 shows the average results which leads to Table 4.23 as prioritization topologies.

A comparison between Table 4.23 and Table 4.15 shows a perfect match between the theoretical results of relative robustness and practical throughput performance for random graphs which proves the effect of the new term (Relative Robustness) in random topologies performance.

4.10 Epidemic Spreading - Term Analysis

The term Epidemic Spreading (E.C)of a node which is defined as maximum eigenvalue of adjacency matrix of a network graph may refer to criticality of the node which makes information spreading through this node smoother and faster.

The term criticality means that this node would be met more often

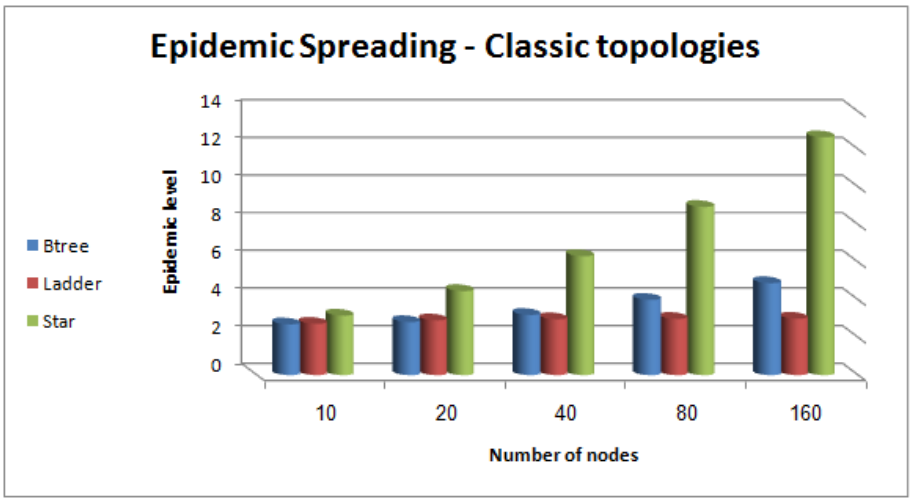

Figure 4.27: Epidemic Spreading - Classic Graphs

randomly than any other nodes in the network. This is used to be the most accessed point of the network.

Thus, having a higher level epidemic spreading node can affect the performance of the network and be considered as a performance feature.

Sometimes this feature is considered as robustness feature as well, but as the characteristics of only one node can't decide the robustness of the whole network, this hypothesis has not been taken here.

4.10.1 E. Spreading -Theoretical analysis of classic graphs

Measuring maximum eigenvalues of adjacency matrices of classic graphs ended up with Figure 4.27 which compares them with each other with different number of nodes. What is noticeable in this graph is the low growth with the ladder graph in comparison with balanced tree and star graphs. As to begin with 10 nodes the epidemic spreading of ladder and balanced tree are almost the same, but further on this feature grows more rapidly for the balanced tree.

Based on the information on Figure 4.27, the prioritization of classic graphs are given in Table 4.16.

Prioritization	Graph
1	Star
2	Ladder
3	Btree

Table 4.16: Classic graphs Theoretical Epidemic Spreading Prioritization

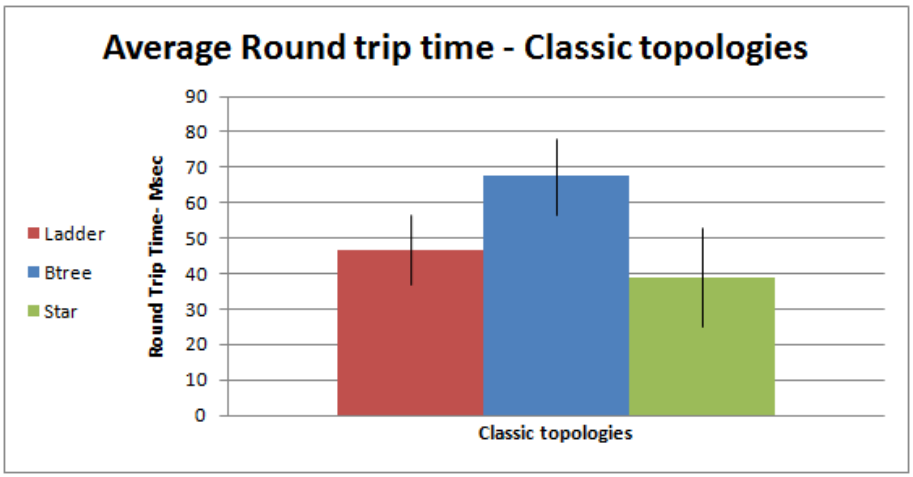

Figure 4.28: Average Round Trip Time - Classic Topologies

4.10.2 E. Spreading -Practical analysis of classic topologies

In order to experiment the epidemic spreading feature a round trip test has been taken. This test is done under stress by sending packets of 2048 bytes at each ping and the average round trip time has been taken for the test. The experiments have been done with 20 times total repetitions and the average round trip time is calculated out of 20 pings each time as well.

In order to have precise results the node with maximum eigenvalue has been included in the path of the pings directly. Figure 4.28 show the results of the experiments for classic topologies.

Based on the results of average round trip time in milliseconds, Table 4.17 shows the prioritization of classic topologies in practice. Note that there is a reverse relationship between topology prioritization and average round trip time as the lower the round trip time is the higher priority the topology has.

Prioritization	Graph
1	Star
2	Ladder
3	Btree

Table 4.17: Classic topologies Practical Epidemic Spreading Prioritization

Comparing Table 4.17 and Table 4.16 shows a perfect match between the results for theoretical and practical epidemic spreading for classic graphs.

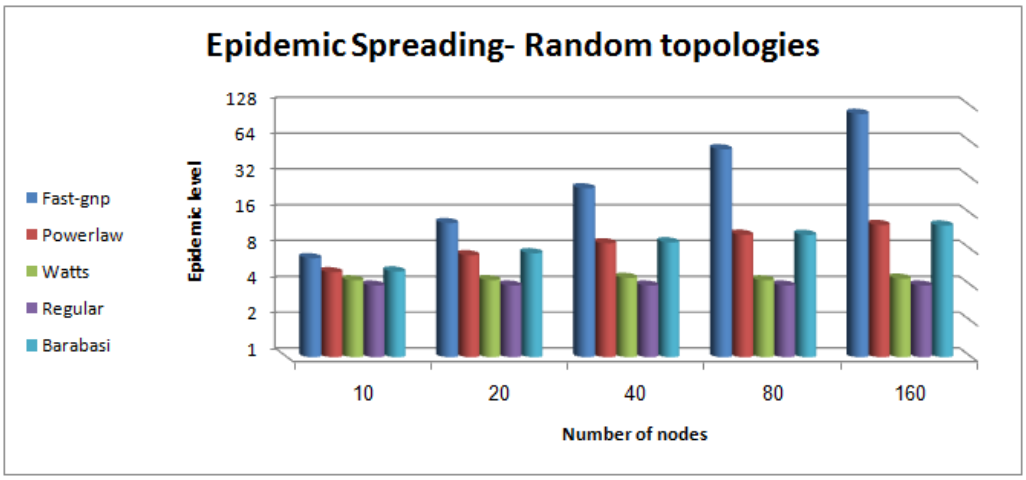

Figure 4.29: Epidemic Spreading - Random Graphs

4.10.3 E. Spreading -Theoretical analysis of random graphs

Considering the maximum eigenvalue of the adjacency matrix of all random graph, Figure 4.29 shows precise comparison between each of the random graphs at each of the scalings.

The differences get more obvious while scaling up and that is because some graphs grow more rapidly than others in scaling. Table 4.18 show the prioritization between these five random graphs in theory considering their maximum eigenvalues.

Prioritization	Graph
1	Fast-gnp
2	Powerlaw
3	Barabasi
4	Watts
5	Regular

Table 4.18: Random graphs Epidemic Spreading Prioritization

4.10.4 E. Spreading -Practical analysis of random topologies

The same experiments of round trip time with large packets have been done for random topologies in practice. Figure 4.30 shows the results for average round trip time for a network of 10 nodes of each of the random topologies.

Table 4.19 shows the prioritization of random topologies that is extracted from the practical experiments. Again note that there is a reverse

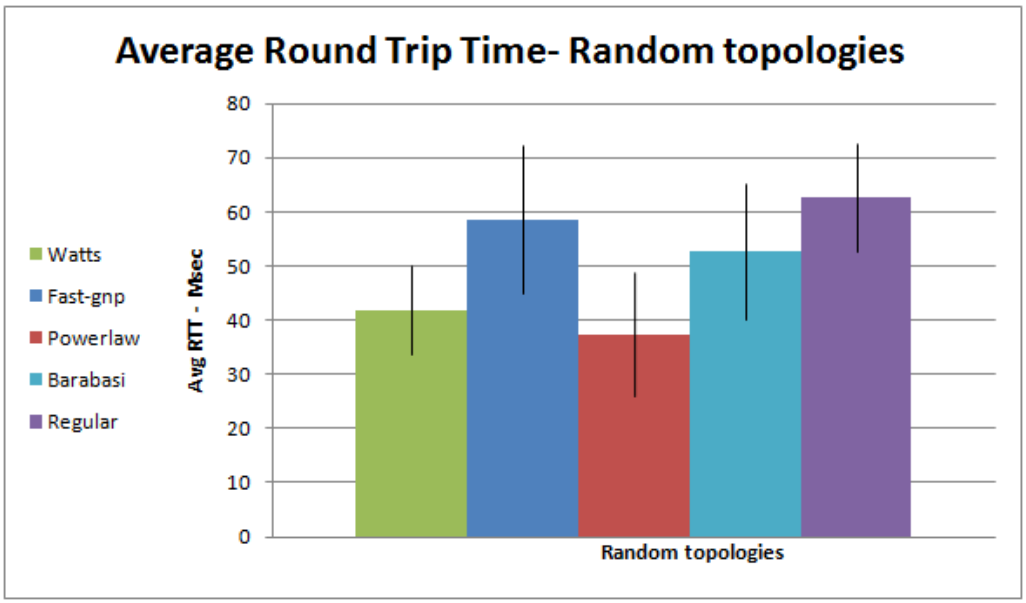

Figure 4.30: Average Round Trip Time - Random Topologies

relationship between topology prioritization and average round trip time as the lower the round trip time is the higher priority the topology has.

Prioritization	Graph
1	Powerlaw
2	Watts
3	Barabasi
4	Fast-gnp
5	Regular

Table 4.19: Random graphs Epidemic Spreading Prioritization

A comparison between Table 4.19 and Table 4.18 shows a mismatch between the calculated results and the results in practice for the epidemic spreading of random topologies.

4.11 Betweenness Centrality - Term Analysis

The term Betweenness Centrality is a stronger measurement for a node in network than just a centrality (epidemic spreading) as it considers the number of all shortest paths which pass through the measured node. Thus, it considers how much load the node can pass through and consequently how important the node is.

The fact is how this feature deal with classic and random topologies and

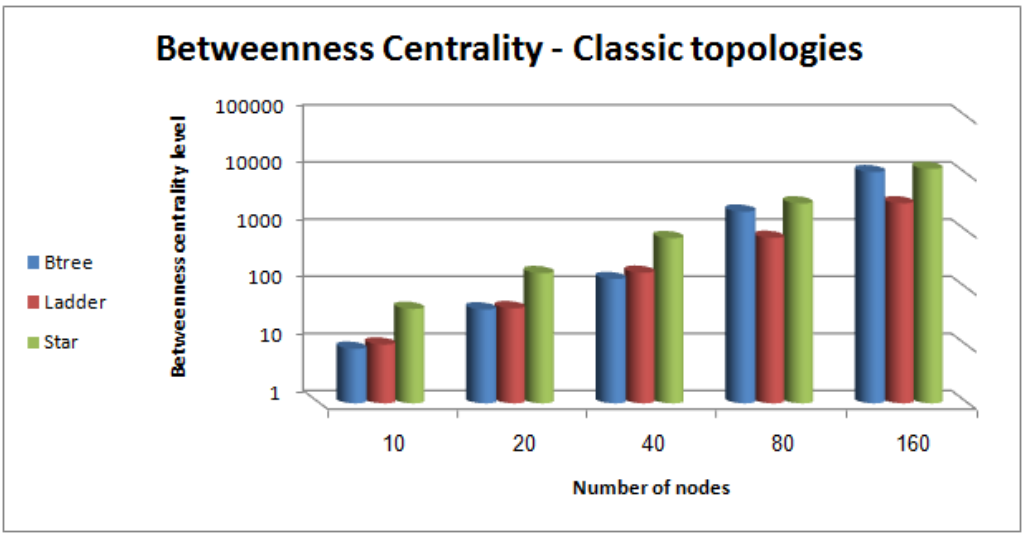

Figure 4.31: Betweenness Centrality - Classic Graphs

would it be a consistent solution to take? Which tests are more reliable within this feature can be another key question to ask as well.

This feature is measured mathematically in this thesis by the powerful function of BetweennessCentrality in Mathematica which gets a graph g as the input and calculates the shortest path betweenness centrality for every single node of the graph.

4.11.1 B. Centrality -Theoretical analysis of classic graphs

For each of the nodes of classic graphs in Mathematica, the function BetweennessCentrality has calculated all shortest paths that have passed through and through a sort function, the nodes with the highest betweeness centrality have been extracted for each type of the graphs.

Figure 4.31 shows the results based on the described calculations and compares classic graphs based on the highest betweenness centrality they have.

what is interesting about the trend of classic graphs in Figure 4.31, is the balanced tree trend which grows faster than two other graphs while scaling up. Thus, balanced tree behaves better in terms of betweenness centrality while scaling up as to be begin with it is comparable with ladder and star has the highest betweenness centrality while at n=160 balanced tree is more comparable to star than ladder which makes the predictions a bit hard in this case.

It is also good to remind that balanced tree at n=10 does not cover 10 nodes

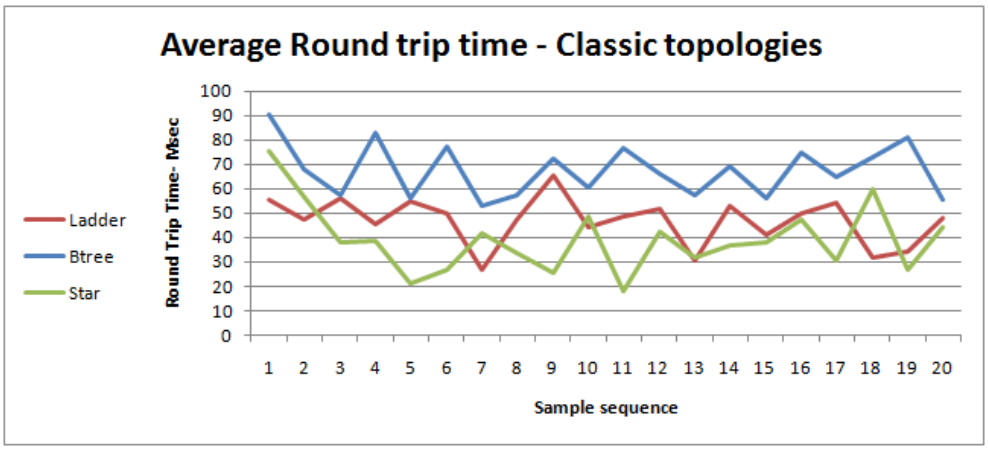

Figure 4.32: Round Trip Time Trend - Classic Topologies

and has only 7 nodes which makes it perform a bit lower than ladder at n=10 most of times.

Table 4.20 shows the prioritization of classic graphs based on the information from Figure 4.31.

Prioritization	Graph
1	Star
2	Ladder
3	Btree

Table 4.20: Classic graphs Betweenness Centrality Prioritization

4.11.2 B. Centrality -Practical analysis of classic topologies

In order to test the betweenness centrality feature, a round trip test has been taken and that is because this is a feature of a single node which can affect the network responses.

As before the test is done under stress by sending packets of 2048 bytes at each ping and the average round trip time has been taken. The experiments have been done with 20 repetitions and the average round trip time is calculated out of 20 pings each time as well.

In order to have precise results, the node with the highest betweenness centrality has been included in the path of the pings directly.

Figure 4.32 shows the results of the experiments on classic graphs.

Extracting average round trip times of the graphs, Table 4.21 shows the prioritization of the graphs.

Prioritization	Graph
1	Star
2	Ladder
3	Btree

Table 4.21: Classic graphs Round Trip Time Prioritization

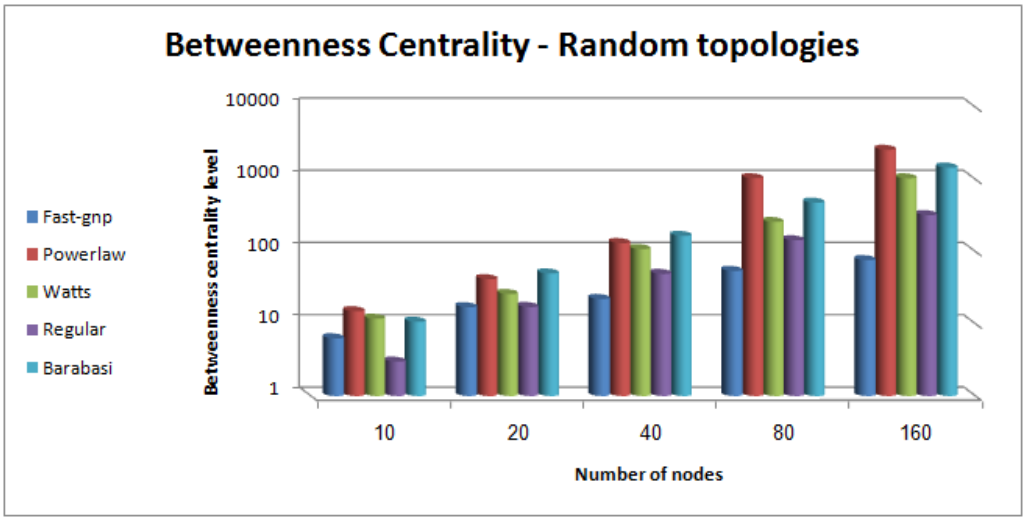

Figure 4.33: Betweenness Centrality - Random Graphs

Comparing Table 4.21 and Table 4.20 shows a perfect match which proves that betweenness centrality can be a good feature to consider the importance of a node in classic graphs.

4.11.3 B. Centrality -Theoretical analysis of random graphs

Calculating betweenness centrality on Mathematica and considering the graphs with the highest betweenness, Figure 4.33 shows the theoretical comparison between the random graphs.

What is interesting about this graph is that the pattern for some of the graphs would change through the scaling. This change can especially be seen in Regular graph which has less betweenness than Fast-gnp in lower scales but this pattern changes completely in higher scales so that it has quite more betweenness in comparison with fast-gnp while n=160.

According to Figure 4.33 the prioritization of random graphs for n=10 in summarized in Table 4.22 as follows:

Prioritization	Graph
1	Powerlaw
2	Watts
3	Barabasi
4	Fast-gnp
5	Regular

Table 4.22: Random graphs Betweenness Centrality Prioritization

4.11.4 B. Centrality -Practical analysis of random graphs

The round trip time experiments with large packets have been done for random topologies as before in practice. Figure 4.30 in a section before showed the results for average round trip time for a network of 10 nodes of each of the random topologies.

A comparison between Table 4.22 and Table 4.19 in previous section which is in fact a comparison between a theoretical analysis of betweenness centrality and practical analysis of round trip time around the betweenness node shows perfect match within random graphs in this case.

Chapter 5

Discussion

In this chapter,the theoretical and practical analysis from previous chapter would be discussed carefully and suitable formulas would be extracted from the calculations for each of the classic and random topologies. Any possible predictions would be discussed carefully as well.

5.1 Algebraic Connectivity

According to results and analysis in the previous chapter ***Algebraic connectivity*** shows perfect results both in theory and practice on classic topologies. All results are taken out of acceptable number of experiment repetitions and can be popularized through other classic topologies both theoretically and practically.

On the other hand, algebraic connectivity feature does not match the characteristics of random topologies as it did on classic topologies. Thus, algebraic connectivity is not a perfect element to predict random topology performance behavior as theoretical and practical results did not match.

A prediction for Ladder and Balanced tree classic topologies based on the connectivity level in Figure 4.20 could be that the performance would decrease at each time scaling up the topologies. Having an almost constant performance with Star topology while scaling up, could be a matter of further experiments.

Comparison between classic and random practical results shows that random Regular and Fast-gnp topologies give better performance results (797 and 766 Mbit/sec) than the best classic topology which is Star (759 Mbit/sec).

But there is no doubt that Star, as a flat topology has a very good performance in small to medium size networks. It is performed even better than some of other random topologies and is standing in third rank among 8 classic and random topologies.

5.2 Relative Robustness

Relative Robustness shows perfect results both in theory and practice on random topologies. Since all the results are taken from acceptable number of experiment repetitions, they can be popularized through other random topologies both theoretically and practically.

The fact is although other elements such as the number of minimum and average node degrees could also affect the the performance and considering them would make the formula more complicated, the current results are still quite trustable in the tested random topologies. Thus, there was no requirement to consider them in the current situation.

On the other hand, relative robustness feature does not match the characteristics of classic topologies as it did on random ones. Thus, this feature is not a perfect element to predict classic topology performance behavior as theoretical and practical results did not perfectly match.

A prediction for Random Regular topology could be that the relative robustness for any number of edges would always be 1 as minimum and average node degree are always the same in this type of random topology. Having almost the same level of performance while scaling up due to the previous fact is open to experiment.

All other random graphs would have a slightly similar relative robustness while scaling up. The differences are mostly due to randomized algorithms which make different graphs each time at the scale up.

5.2.1 Relative Robustness Discussion Considerations

Defining such fraction for random graphs might bring questions in mind. Below are main categories that the author of the thesis has thought of while defining such variable:

- Consistency I - The first fact is that the random graphs of the experiments in this thesis were well connected graphs by the help of high probability and average node degrees. Supposing any type of disconnected random graph, the minimum node degree would drop

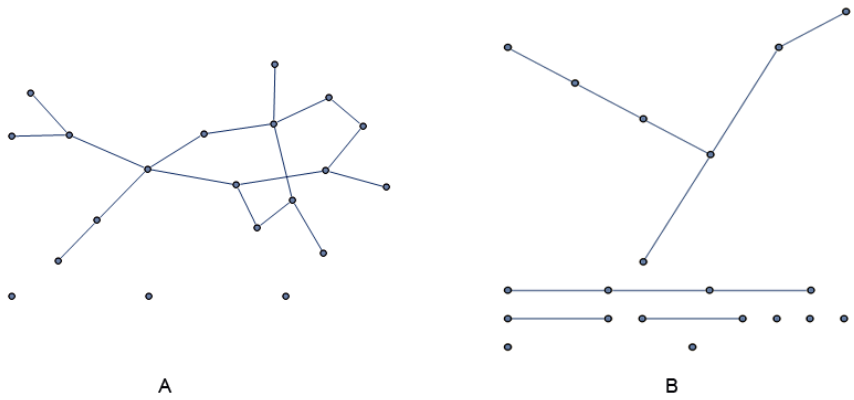

Figure 5.1: Disconnected graph with (A)3 min node degree and (B)5 min node degree

to zero.

Figure 5.1 shows two Fast-gnp graphs with 20 nodes that are generated with exactly the same algorithm but in two times randomly. Each time has made a disconnected graph with different number of unconnected nodes. What is obvious is that both graphs have the same min node degree of zero which makes the fraction of Relative Robustness equal to zero. In this case, Relative Robustness can not talk about the robustness of the graphs and compare which one is better connected although graph A has a better average node degree than graph B.

Thus, Relative Robustness is valid for connected random graphs and the evidence for that could be a positive second smallest laplacian eigenvalue. This can apply to a connected data center in real world as an example.

- Consistency II - As Figure 5.2 shows, a random graph can be disconnected (as in A) but with minimum degree of greater than zero. This implies that the defined fraction for Relative Robustness does not get zero but it might end up with wrong information about the robustness of the graph.

  As Figure 5.2 shows both disconnected graph A and connected graph B have the same average node degree of 1.6 and minimum node degree of one while first one has 10 nodes and second one consists of 6 nodes. Anyhow Relative Robustness fraction is going to give the same result for both of the graphs. Supposing robustness to have a path between every two nodes, first graph fails. On the other hand measuring throughput between the connected nodes, one can't get the the predicted results by the defined fraction.

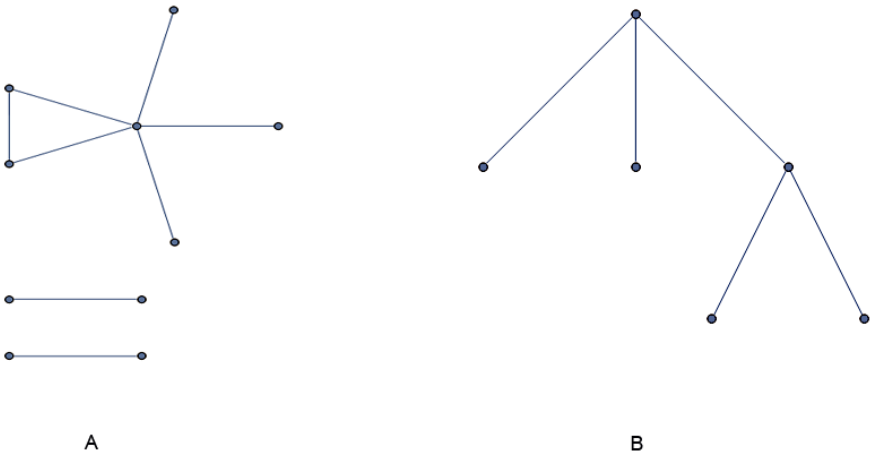

Figure 5.2: (A)Disconnected graph and (B) Connected graph, both with the same average and minimum node degrees.

Thus, here is another point of failure which implies that this definition works the best with a well connected graph. Although there is a paradox in the definition of robustness in a disconnected graph as there is no single path between every two nodes of such graph.

- Consistency III - On the other hand, when minimum node degree is as big as the average degree the fraction equals to 1. This has happened in Random Regular graphs where minimum node degree equals the average node degree. This is the highest level of Relative Robustness and the fraction is consistent in such situations.

- Improvement - Although in order to be precise enough, it was possible to consider the number of minimum node degrees or even second minimum node degrees, the experiments about the chosen random graphs showed the algorithms to make the graphs are far enough not to make too close average and minimum node degrees so that the number of minimum node degrees become an important factor.

  On the other hand as mentioned earlier, the number of minimum node degrees can not go high up as the average node degree would be affected as well.

  Thus, in case of the random graphs in the perspective of this thesis which is quite a wide perspective in range of random graphs as well, the formula for Relative Robustness for well-connected random graphs seemed consistent and the author did not go for applying complicated changes to improve the formula.

5.3 Epidemic Spreading

Epidemic Spreading shows perfect results both in theory and practice only on classic topologies. As all the results are taken from acceptable number of experiment repetitions, they can be popularized through other classic topologies both theoretically and practically.

Thus, maximum eigenvalue of adjacency matrix of a classic network graph can be a feature that affects the centrality and experiments such as ping and variables such as round trip times.

On the other hand, epidemic spreading feature does not match the characteristics of random topologies as it did on classic ones. Thus, this feature is not a perfect element to predict random topology performance behavior as theoretical and practical results did not perfectly match. The result could be predictable though as the shortest paths algorithms are more complicated in random graphs, thus centrality could not be a perfect feature for such graphs.

A prediction for Star topology could be that the gap between the round trip time of Star topology and other two topologies gets bigger while scaling in a way that Star would show quite smaller RT times in milliseconds in comparison with other two topologies.

As the second prediction and by a smaller level of growth, Balanced tree would also show smaller RT times than the Ladder while scaling up the networks.

5.4 Betweenness Centrality

Betweenness Centrality shows perfect results in theory and practice both on classic and random topologies. As all the results are taken from acceptable number of experiment repetitions, they can be popularized through other classic and random topologies both theoretically and practically.

Thus, Betweenness Centrality of a network graph can be a feature that affects the experiments such as ping and variables such as round trip times.

Unlike the epidemic spreading which is only valid for classic topologies, betweenness centrality is valid for both graph topologies. Thus, betweenness centrality can be a feature to substitute the epidemic spreading when both classic and random topologies are going to be examined.

A prediction for Balanced tree topology could be that the gap between the round trip time of Balance tree topology and other two topologies gets smaller while scaling in a way that Balanced tree would show quite smaller RT times in milliseconds in comparison especially with ladder topology.

As the second prediction for random graphs, the round trip time for Regular topology would get smaller (better) in milliseconds as the graphs scale up in comparison with Fast-gnp graph in specific.

5.5 Overall Discussion

Based on the results on the previous chapter, not all the defined variables are suitable for all types of network topologies.

generally, the defined variables can be divided into two categories of performance and node importance which have global and local network effects respectively. This means with performance variables we can expect more global effects and with node importance variables the effects are specialized more local to the important node(s). Each category can also affect one or two type of network topologies as follow in Table 5.1 and Table 5.2.

Performance feature	Supported topology
Algebraic Connectivity	Classic
Relative Robustness	Random

Table 5.1: Performance features and corresponding network topology types

What this thesis tried to figure out was a better way of dealing with performance of random topologies. The author believed that Relative Robustness would affect the performance of random graphs better than just the average degree and the thesis proved this belief with sets of experiments as well.

Node importance feature	Supported topology
Epidemic Spreading	Classic
Betweenness Centrality	Classic-Random

Table 5.2: Node importance and corresponding network topology types

On the other hand, Betweenness Centrality has been shown to be a stronger feature in comparison with Epidemic spreading which affects the node importance in both classic and random topologies. Therefore, in similar

cases measuring only the betweenness centrality can save more time and is still trustworthy enough.

Although any prediction can be an open experiment, they can be trustworthy to some extent and further are the predictions that can be extracted from the results and analysis of this thesis.

Based on features on Table 5.1 a prediction for the throughput of classic topologies in case of algebraic connectivity would be categorized as below while scaling up the network:

- Star
- Ladder
- Balanced Tree

The same prediction and categorization can be taken from Table 5.1 for the throughput of random topologies in case of relative robustness while scaling up the network as below:

- Regular
- Fast-gnp
- Barabasi
- Watts
- Powerlaw

The only prediction which the author will definitely experiment as the next step is the prediction which says that *the relative robustness of the regular and watts random topologies would stay constant while scaling up* the network. This can be tested by measuring the throughput of the mentioned topologies while scaling the networks up. This would imply that as the minimum and average node degrees are constant and predefined in these two topologies, relative robustness stays constant as well. This might get affected by some network features which we have not considered. Thus, testing such predictions is recommended.

Based on Table 5.2 a prediction for the node importance of classic topologies in case of epidemic spreading can be that star topology has the highest importance node in general, but balanced tree shows better importance node features in contrast with ladder topology while scaling up to higher node numbers.

Table 5.2 also gives a prediction and categorization pattern for the

node importance of random topologies in case of betweenness centrality which has more local effects than the global ones. Results are listed as below:

- Powerlaw
- Watts
- Barabasi
- Fast-gnp
- Regular

At the same time testbed can affect the analysis a lot, although it does not make any part of the topologies. The centralized loop avoidance of the Openflow did not handle the complex loops in random graphs.

This does not mean that Openflow in general can not handle such a scenario as in some integrated versions of Openflow, where the loop avoidance is not centralized, it would be done within the switches and this problem would probably not occur. But centralized Openflow with Mininet simulation failed the experiment at least.

Chapter 6

Future Work and Conclusion

6.1 Future Work

As this thesis ended up with a great amount of data and some predictions and also a difficulty in testbed, it can be considered as a very good opportunity for further future work in terms of another thesis or other smaller projects or research papers.

Below some of the topics would be discussed briefly to describe the situation and inspire the motivation:

6.1.1 Loop avoidance

Openflow platform with the centralized loop avoidance mechanism did not succeed in avoiding loops of random topologies. Therefore, there are three main mechanisms that could be taken as future work to fix this problem.

1. Non-centralized loop avoidance solution - This could be a switch based STP solution which takes the decision of loop avoidance from the controller and give it back to OpenVswitches for instance. If the loop avoidance problem was due to controller based decision making and the complexity of the decisions multiple transfers between the switches and the controller, this could be a solution.

2. Trying other loop avoidance protocols - This can be any new loop avoidance mechanism which is customized for Openflow or basically a feasibility and performance study of other protocols such as SPB (Shortest Path Bridging), MC-LAG (Multi-Chassis Link Aggregation),

etc.

3. Networkx virtual environment compatibility - To work with Mininet or any other virtualization environment and creating random topologies by the help of Networkx and python programming, there could be a direct way of importing graphs into Mininet custom topologies.

   Now, it seems some bugs exists in this way within a 32 bits Mininet virtual machine. With this feature working properly, the mentioned problem would be solved in case of virtual environment and created graphs with applied loop avoidance features can be imported directly into Mininet as custom topologies. This is how it is done now in an undirect way in this thesis, although this does not let the Openflow apply its loop avoidance algorithms within the controller or its switches.

6.1.2 Predictions

Predictions are open to experiment, but some seem more important to test. Among all the predictions that are extracted from different phases of this thesis there are two main categories that can be a subject to good researches. Below is a brief description of them:

1. Relative Robustness - This prediction needs a definite experiment as it says that the relative robustness of the random regular and watts random topologies would stay constant within scaling up the network. The tests can be done by measuring the throughput of the mentioned topologies and scale up the networks.

   If it works properly it might imply that as the minimum and average node degrees are predefined in these topologies, relative robustness stays constant in the experiments as well. On the other hand this might get affected by some other features which we have not been considered. Therefore, testing such predictions is recommended.

2. Node Importance - A prediction for the node importance of classic topologies in case of *epidemic spreading* can be that star topology has the highest importance node, but balanced tree shows better importance node features than ladder topology in scaling to higher node numbers. This is also a subject to experiment, measure and compare both theoretical and practical features.

6.1.3 Percolation

In case of studying random graphs properties, specially large infinite ones, percolation theory can be an interesting future work. It is basically related to robustness part of the thesis and the average degree of the random graphs in very large scales. It probes how robust and connected the graphs could be while removing fractions of 1-p of the graph G with the probability distribution of p. This part has not been done due to limitations of implementing such large-scale networks. This could be done all theoretically.

6.2 Conclusion

The main idea of problem statement of this thesis was to find out how a variety of classic and random network topologies perform on an Openflow testbed, while trying to put more effort on differentiating measurement variables between classic and random graphs. It was also trying to predict to a certain degree in order to categorize the topologies and leave the rest for further experiments and future work.

Thus, below are conclusion items that are related to the corresponding problem statements questions:

1. *What features can be used as variables to measure connectivity, robustness, spreading and centrality of a classic and random network graph?*

2. *What is the behavior of network topologies in case of connectivity, robustness, spreading and centrality variables?*

    - Measuring Algebraic Connectivity to show how well the graph is connected, is defined by second minimum laplacian eigenvalue and it is shown that it is supported by classic topologies in practice.

    - Relative Robustness is a new variable defined specifically in this thesis by minimum node degree divided by average node degree to show accurately how robust a topology is and it is shown that it is supported by random topologies in practice.

    - Measuring Epidemic Spreading to show criticality of a node which makes information spread through it smoother, is defined with maximum eigenvalue of adjacency matrix and it is shown that it is supported by classic topologies in practice.

- Measuring Betweenness Centrality to consider how much load a node can pass through, is defined by BetweennessCentrality function of Mathematica and it is shown that it is supported by both classic and random topologies and it is a stronger node importance feature.

3. *What can be a prediction and categorization for the classic and random network graphs while scaling up?*

   Generally classic topologies are better subjects to predictions and it is possible to predict them by numbers. But with random topologies, the probability distribution makes it more unexpected to predict although these topologies show more robustness through scaling as the level of strength in different defined variables remains fixed or changes insignificantly most of times.

   Partial predictions are given in discussion section of each variable in Analysis and Discussion chapter. Referring to Table 5.1 and Table 5.2 representing throughput and node importance feature variables Table 6.1 gives the overview of all topologies categorization as a conclusion. As mentioned before the difference between performance features and node importance features can be their effects on the network in terms of global and local.

Classic (Perf-Node imp)	Random (Perf)	Random (Node imp)
Star	Regular	Powerlaw
Ladder	Fast-gnp	Watts
B-tree	Barabasi	Barabasi
	Watts	Fast-gnp
	Pwerlaw	Regular

Table 6.1: Classic and Random topologies performance and node importance categorization

4. Openflow as the testbed of this thesis affected the results by its centralized loop avoidance technology. In real world, such network scenarios with complex loops of random graphs should be handled with integrated technologies in which, network switches take the responsibility of loop avoidance instead of the centralized loop avoidance of Openflow did not quite manage the scenario and some corrections which mentioned in previous chapter had to be taken.

Bibliography

[1] Wireshark the world's foremost network protocol analyzer @ONLINE. Accessed Jan, 2013.

[2] An introduction to quantum computing for non-physicists. *ACM Comput. Surv.*, 32(3):300–335, September 2000.

[3] Floodlight Admin. Cbench @ONLINE, Mar 2012. Accessed Feb, 2013.

[4] Edoardo M. Airoldi and Kathleen M. Carley. Sampling algorithms for pure network topologies: a study on the stability and the separability of metric embeddings. *SIGKDD Explor. Newsl.*, 7(2):13–22, December 2005.

[5] Ehab Al-Shaer and Saeed Al-Haj. Flowchecker: configuration analysis and verification of federated openflow infrastructures. In *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration*, SafeConfig '10, pages 37–44, New York, NY, USA, 2010. ACM.

[6] Murphy McCauley Ali Al-Shabibi. Pox wiki @ONLINE, February 2013. Accessed Jan, 2013.

[7] Muhammad Bilal Anwer, Murtaza Motiwala, Mukarram bin Tariq, and Nick Feamster. Switchblade: a platform for rapid deployment of network protocols on programmable hardware. *SIGCOMM Comput. Commun. Rev.*, 40(4):183–194, August 2010.

[8] Hirochika Asai, Kensuke Fukuda, and Hiroshi Esaki. Traffic causality graphs: profiling network applications through temporal and spatial causality of flows. In *Proceedings of the 23rd International Teletraffic Congress*, ITC '11, pages 95–102. ITCP, 2011.

[9] Petra Berenbrink, Colin Cooper, and Tom Friedetzky. Random walks which prefer unvisited edges.: exploring high girth even degree expanders in linear time. In *Proceedings of the 2012 ACM symposium on Principles of distributed computing*, PODC '12, pages 29–36, New York, NY, USA, 2012. ACM.

[10] Alireza Bigdeli, Ali Tizghadam, and Alberto Leon-Garcia. Comparison of network criticality, algebraic connectivity, and other graph metrics. In *Proceedings of the 1st Annual Workshop on Simplifying Complex Network for Practitioners*, SIMPLEX '09, pages 4:1–4:6, New York, NY, USA, 2009. ACM.

[11] Gemma Boleda, Eva Maria Vecchi, Miquel Cornudella, and Louise McNally. First-order vs. higher-order modification in distributional semantics. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, EMNLP-CoNLL '12, pages 1223–1233, Stroudsburg, PA, USA, 2012. Association for Computational Linguistics.

[12] Anat Bremler-Barr, David Hay, Danny Hendler, and Ron M. Roth. Peds: a parallel error detection scheme for tcam devices. *IEEE/ACM Trans. Netw.*, 18(5):1665–1675, October 2010.

[13] Mark Burgess. *Analytical Network and System Administration*. Wiley, 1st edition, April 2004.

[14] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. Devoflow: scaling flow management for high-performance networks. *SIGCOMM Comput. Commun. Rev.*, 41(4):254–265, August 2011.

[15] NetworkX Developers. *NetworkX documentation*, April 2012.

[16] Shlomi Dolev, Yuval Elovici, and Rami Puzis. Routing betweenness centrality. *J. ACM*, 57(4):25:1–25:27, May 2010.

[17] Omar El Ferkouss, Ilyas Snaiki, Omar Mounaouar, Hamza Dahmouni, Racha Ben Ali, Yves Lemieux, and Cherkaoui Omar. A 100gig network processor platform for openflow. In *Proceedings of the 7th International Conference on Network and Services Management*, CNSM '11, pages 286–289, Laxenburg, Austria, Austria, 2011. International Federation for Information Processing.

[18] Omar El Ferkouss, Ilyas Snaiki, Omar Mounaouar, Hamza Dahmouni, Racha Ben Ali, Yves Lemieux, and Cherkaoui Omar. A 100gig network processor platform for openflow. In *Proceedings of the 7th International Conference on Network and Services Management*, CNSM '11, pages 286–289, Laxenburg, Austria, Austria, 2011. International Federation for Information Processing.

[19] Greg Ferro. Controller ecosystem. Webinar, 2012. Accessed Feb, 2013.

[20] Greg Ferro. Northbound api, southbound api, east/north lan navigation in an openflow world and an sdn compass @ONLINE,

August 2012. Accessed Feb, 2013.

[21] Airton Ishimori (GERCOM/UFPA), Fernando Faria (GERCOM/UFPA), Igor Carvalho (GERCOM/UFPA), Eduardo Cerqueira, and Antonio Abelem (GERCOM/UFPA). Automatic qos management on openflow software-defined networks, June 2012.

[22] Ran Giladi and Niv Yemini. A programmable, generic forwarding element approach for dynamic network functionality. In *Proceedings of the 2nd ACM SIGCOMM workshop on Programmable routers for extensible services of tomorrow*, PRESTO '09, pages 19–24, New York, NY, USA, 2009. ACM.

[23] Debra S. Goldberg, Dirk Grunwald, Clayton Lewis, Jessica A. Feld, and Sarah Hug. Engaging computer science in traditional education: the ecsite project. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*, ITiCSE '12, pages 351–356, New York, NY, USA, 2012. ACM.

[24] Greg Goth. Software-defined networking could shake up more than packets. *IEEE Internet Computing*, 15(4):6–9, July 2011.

[25] Amit Goyal, Hal Daumé, III, and Raul Guerra. Fast large-scale approximate graph construction for nlp. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, EMNLP-CoNLL '12, pages 1069–1080, Stroudsburg, PA, USA, 2012. Association for Computational Linguistics.

[26] Fang Hao, T. V. Lakshman, Sarit Mukherjee, and Haoyu Song. Enhancing dynamic cloud-based services using network virtualization. *SIGCOMM Comput. Commun. Rev.*, 40(1):67–74, January 2010.

[27] Brandon Heller, Rob Sherwood, and Nick McKeown. The controller placement problem. In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, pages 7–12, New York, NY, USA, 2012. ACM.

[28] Michael Jarschel, Simon Oechsner, Daniel Schlosser, Rastin Pries, Sebastian Goll, and Phuoc Tran-Gia. Modeling and performance evaluation of an openflow architecture. In *Proceedings of the 23rd International Teletraffic Congress*, ITC '11, pages 1–7. ITCP, 2011.

[29] Brent ByungHoon Kang, Eric Chan-Tin, Christopher P. Lee, James Tyra, Hun Jeong Kang, Chris Nunnery, Zachariah Wadler, Greg Sinclair, Nicholas Hopper, David Dagon, and Yongdae Kim. Towards complete node enumeration in a peer-to-peer botnet. In *Proceedings of the 4th International Symposium on Information, Computer, and Com-*

munications Security, ASIACCS '09, pages 23–34, New York, NY, USA, 2009. ACM.

[30] Eric Keller, Soudeh Ghorbani, Matt Caesar, and Jennifer Rexford. Live migration of an entire network (and its hosts). In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, HotNets-XI, pages 109–114, New York, NY, USA, 2012. ACM.

[31] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: verifying network-wide invariants in real time. *SIGCOMM Comput. Commun. Rev.*, 42(4):467–472, September 2012.

[32] Bong Jun Ko, Vasileios Pappas, Ramya Raghavendra, Yang Song, Raheleh B. Dilmaghani, Kang-won Lee, and Dinesh Verma. An information-centric architecture for data center networks. In *Proceedings of the second edition of the ICN workshop on Information-centric networking*, ICN '12, pages 79–84, New York, NY, USA, 2012. ACM.

[33] Darren Lim. Taking students out for a ride: using a board game to teach graph theory. *SIGCSE Bull.*, 39(1):367–371, March 2007.

[34] Thomas A. Limoncelli. Openflow: a radical new idea in networking. *Commun. ACM*, 55(8):42–47, August 2012.

[35] Thomas A. Limoncelli. Openflow: a radical new idea in networking. *Commun. ACM*, 55(8):42–47, August 2012.

[36] Thomas A. Limoncelli. Openflow: a radical new idea in networking. *Commun. ACM*, 55(8):42–47, August 2012.

[37] Thomas A. Limoncelli. Openflow: a radical new idea in networking. *Commun. ACM*, 55(8):42–47, August 2012.

[38] Pingping Lin, Jun Bi, and Hongyu Hu. Asic: an architecture for scalable intra-domain control in openflow. In *Proceedings of the 7th International Conference on Future Internet Technologies*, CFI '12, pages 21–26, New York, NY, USA, 2012. ACM.

[39] Diego R. Lopez. Applying abstraction for a more efficient and fair network usage. In *Proceedings of the 2012 ACM workshop on Capacity sharing*, CSWS '12, pages 1–2, New York, NY, USA, 2012. ACM.

[40] Kathy Macropol and Ambuj Singh. Scalable discovery of best clusters on large graphs. *Proc. VLDB Endow.*, 3(1-2):693–702, September 2010.

[41] Jon Matias, Borja Tornero, Alaitz Mendiola, Eduardo Jacob, and Nerea Toledo. Implementing layer 2 network virtualization using openflow: Challenges and solutions. In *Proceedings of the 2012 European Workshop*

on Software Defined Networking, EWSDN '12, pages 30–35, Washington, DC, USA, 2012. IEEE Computer Society.

[42] Rick McGeer. A safe, efficient update protocol for openflow networks. In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, pages 61–66, New York, NY, USA, 2012. ACM.

[43] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[44] Marc Mendonca, Katia Obraczka, and Thierry Turletti. The case for software-defined networking in heterogeneous networked environments. In *Proceedings of the 2012 ACM conference on CoNEXT student workshop*, CoNEXT Student '12, pages 59–60, New York, NY, USA, 2012. ACM.

[45] Farnaz Moradi, Magnus Almgren, Wolfgang John, Tomas Olovsson, and Philippas Tsigas. On collection of large-scale multi-purpose datasets on internet backbone links. In *Proceedings of the First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*, BADGERS '11, pages 62–69, New York, NY, USA, 2011. ACM.

[46] Jad Naous, David Erickson, G. Adam Covington, Guido Appenzeller, and Nick McKeown. Implementing an openflow switch on the netfpga platform. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '08, pages 1–9, New York, NY, USA, 2008. ACM.

[47] Suman Nath, Zachary Anderson, and Srinivasan Seshan. Choosing beacon periods to improve response times for wireless http clients. In *Proceedings of the second international workshop on Mobility management & wireless access protocols*, MobiWac '04, pages 43–50, New York, NY, USA, 2004. ACM.

[48] Joshua O'Madadhain, Jon Hutchins, and Padhraic Smyth. Prediction and ranking algorithms for event-based network data. *SIGKDD Explor. Newsl.*, 7(2):23–30, December 2005.

[49] Open Networking Foundation. *OpenFlow Switch Specification*, 1.3.0 edition, June 2012.

[50] Alexander Clemm Ralf Wolter. *Network-Embedded Management and Applications: Understanding Programmable Networking Infrastructure*. Springer, 2013 edition, 2012.

[51] Julien Ridoux, Anne Fladenmuller, Yannis Viniotis, and Kavé Sala-

matian. Trellis-based virtual regular addressing structures in self-organized networks. In *Proceedings of the 4th IFIP-TC6 international conference on Networking Technologies, Services, and Protocols; Performance of Computer and Communication Networks; Mobile and Wireless Communication Systems*, NETWORKING'05, pages 511–522, Berlin, Heidelberg, 2005. Springer-Verlag.

[52] Elisa Rojas and Guillermo Ibañez. Torii hlmac: distributed, fault-tolerant, zero configuration data center architecture with multiple tree-based addressing and forwarding. In *Proceedings of The ACM CoNEXT Student Workshop*, CoNEXT '11 Student, pages 25:1–25:2, New York, NY, USA, 2011. ACM.

[53] Nadi Sarrar, Steve Uhlig, Anja Feldmann, Rob Sherwood, and Xin Huang. Leveraging zipf's law for traffic offloading. *SIGCOMM Comput. Commun. Rev.*, 42(1):16–22, January 2012.

[54] Vyas Sekar, Michael K. Reiter, and Hui Zhang. Revisiting the case for a minimalist approach for network flow monitoring. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, IMC '10, pages 328–341, New York, NY, USA, 2010. ACM.

[55] Rob Sherwood, Glen Gibb, Kok-kiong Yap, Guido Appenzeller, Martin Casado, Nick Mckeown, and Guru Parulkar. Flowvisor : A network virtualization layer flowvisor : A network virtualization layer. *OpenFlow Switch*, page 15, 2009.

[56] Hideyuki Shimonishi, Takashi Yoshikawa, and Atsushi Iwata. Off-the-path flow handling mechanism forhigh-speed and programmable traffic management. In *Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, PRESTO '08, pages 15–20, New York, NY, USA, 2008. ACM.

[57] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. Jellyfish: networking data centers randomly. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 17–17, Berkeley, CA, USA, 2012. USENIX Association.

[58] Brent Stephens, Alan Cox, Wes Felter, Colin Dixon, and John Carter. Past: scalable ethernet for data centers. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, CoNEXT '12, pages 49–60, New York, NY, USA, 2012. ACM.

[59] P. Pan T. Nadeau. Framework for software defined networks draft-nadeau-sdn-framework-01. Ietf draft, IETF, April 2012.

[60] Mininet Team. Mininet overview @ONLINE, 2012. Accessed Jan, 2013.

[61] Amin Tootoonchian and Yashar Ganjali. Hyperflow: a distributed control plane for openflow. In *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, INM/WREN'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.

[62] Alexander Traud, Jürgen Nagler-Ihlein, Frank Kargl, and Michael Weber. Cyclic data synchronization through reusing syncml. In *Proceedings of the The Ninth International Conference on Mobile Data Management*, MDM '08, pages 165–172, Washington, DC, USA, 2008. IEEE Computer Society.

[63] Laurent Vanbever, Stefano Vissicchio, Cristel Pelsser, Pierre Francois, and Olivier Bonaventure. Lossless migrations of link-state igps. *IEEE/ACM Trans. Netw.*, 20(6):1842–1855, December 2012.

[64] Andreas Voellmy, Hyojoon Kim, and Nick Feamster. Procera: a language for high-level reactive network control. In *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, pages 43–48, New York, NY, USA, 2012. ACM.

[65] Kuang-Ching Wang. Floodlight documentation @ONLINE, November 2012. Accessed Feb, 2013.

[66] Richard Wang, Dana Butnariu, and Jennifer Rexford. Openflow-based server load balancing gone wild. In *Proceedings of the 11th USENIX conference on Hot topics in management of internet, cloud, and enterprise networks and services*, Hot-ICE'11, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association.

[67] Yang Wang, Deepayan Chakrabarti, Chenxi Wang, and Christos Faloutsos. Epidemic spreading in real networks: An eigenvalue viewpoint. In *In SRDS*, pages 25–34, 2003.

[68] Eric Weisstein. Eigenvalue @ONLINE, February 2013. Accessed March, 2013.

[69] Eric W. Weisstein. "eigenvalue." from mathworld @ONLINE, May 2013. Accessed May, 2013.

[70] Yanhui Xiao, Zhenfeng Zhu, and Yao Zhao. Correlation preserved dictionary learning for sparse representation. In *Proceedings of the 4th International Conference on Internet Multimedia Computing and Service*, ICIMCS '12, pages 196–199, New York, NY, USA, 2012. ACM.

[71] Kok-Kiong Yap, Sachin Katti, Guru Parulkar, and Nick McKeown. Delivering capacity for the mobile internet by stitching together networks. In *Proceedings of the 2010 ACM workshop on Wireless of the students, by the students, for the students*, S3 '10, pages 41–44, New York,

NY, USA, 2010. ACM.

[72] Kok-Kiong Yap, Sachin Katti, Guru Parulkar, and Nick McKeown. Delivering capacity for the mobile internet by stitching together networks. In *Proceedings of the 2010 ACM workshop on Wireless of the students, by the students, for the students*, S3 '10, pages 41–44, New York, NY, USA, 2010. ACM.

[73] Kok-Kiong Yap, Masayoshi Kobayashi, Rob Sherwood, Te-Yuan Huang, Michael Chan, Nikhil Handigol, and Nick McKeown. Openroads: empowering research in mobile networks. *SIGCOMM Comput. Commun. Rev.*, 40(1):125–126, January 2010.

[74] Kok-Kiong Yap, Masayoshi Kobayashi, David Underhill, Srinivasan Seetharaman, Peyman Kazemian, and Nick McKeown. The stanford openroads deployment. In *Proceedings of the 4th ACM international workshop on Experimental evaluation and characterization*, WINTECH '09, pages 59–66, New York, NY, USA, 2009. ACM.

[75] Kok-Kiong Yap, Rob Sherwood, Masayoshi Kobayashi, Te-Yuan Huang, Michael Chan, Nikhil Handigol, Nick McKeown, and Guru Parulkar. Blueprint for introducing innovation into wireless mobile networks. In *Proceedings of the second ACM SIGCOMM workshop on Virtualized infrastructure systems and architectures*, VISA '10, pages 25–32, New York, NY, USA, 2010. ACM.

[76] Shiyou Zhao, Kaushik Roy, and Cheng Kok Koh. Frequency domain analysis of switching noise on power supply network. In *Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design*, ICCAD '00, pages 487–492, Piscataway, NJ, USA, 2000. IEEE Press.

Appendix A

Scripts and mathematic equations

A.1 Networkx graph script

Listing A.1: graph.py

```
#!/usr/bin/python

import networkx as NX # import networkx
from mininet.topo import Topo

# create a graph using a built-in graph generator from networkx
#barabasi graph
G = NX.barabasi_albert_graph(10, 2, seed=None)

#or balanced tree graph
#G = NX.balanced_tree(8, 2, create_using=None)

#or erdos-renyi graph
#G = NX.erdos_renyi_graph(10, 0.7, seed=None, directed=False)

#or fast-gnp graph
#G = NX.fast_gnp_random_graph(10, 0.7, seed=None,directed=False)

#or ladder graph
#G = NX.ladder_graph(5, create_using=None)

#or random regular graph
#G = NX.random_regular_graph(3, 10, seed=None)

#or star graph
#G = NX.star_graph(10, create_using=None)

#or watts strogatz graph
#G = NX.watts_strogatz_graph(10, 7, 0.7, seed=None)

type(G)

#<class 'networkx.classes.digraph.DiGraph'>
# express the graph as an Adjacency Matrix

AM = NX.to_numpy_matrix(G)
```

```
37
38 # use a built-in function from NumPy
39 # to save the Adjacency Matrix as a text file
40
41 import numpy as NP # import the library
42 NP.savetxt("s2/bar10.txt", AM, delimiter=',', newline="\n", fmt='%d')
```

A.2 Matrix based topology scripts

A.2.1 Barabasi matrix topology script

Listing A.2: barabasi.py

```
1  #!/usr/bin/python
2
3  from mininet.topo import Topo, Node
4
5  class MyTopo( Topo ):
6      "Simple topology example."
7
8      def __init__( self, enable_all = True ):
9          "Create custom topo."
10
11         # Add default members to class.
12         super( MyTopo, self ).__init__()
13
14         # Set Node IDs for hosts and switches
15         s1=1
16         s2=2
17         s3=3
18         s4=4
19         s5=5
20         s6=6
21         s7=7
22         s8=8
23         s9=9
24         s10=10
25         h1=11
26         h2=22
27         h3=33
28         h4=44
29         h5=55
30         h6=66
31         h7=77
32         h8=88
33         h9=99
34         h10=100
35         # Add nodes
36       self.add_node( s1, Node( is_switch=True ) )
37         self.add_node( s2, Node( is_switch=True ) )
38         self.add_node( s3, Node( is_switch=True ) )
39         self.add_node( s4, Node( is_switch=True ) )
40         self.add_node( s5, Node( is_switch=True ) )
41         self.add_node( s6, Node( is_switch=True ) )
42         self.add_node( s7, Node( is_switch=True ) )
43         self.add_node( s8, Node( is_switch=True ) )
44         self.add_node( s9, Node( is_switch=True ) )
45         self.add_node( s10, Node( is_switch=True ) )
46
47         self.add_node( h1, Node( is_switch=False ) )
48         self.add_node( h2, Node( is_switch=False ) )
```

```
            self.add_node( h3, Node( is_switch=False ) )
            self.add_node( h4, Node( is_switch=False ) )
            self.add_node( h5, Node( is_switch=False ) )
            self.add_node( h6, Node( is_switch=False ) )
            self.add_node( h7, Node( is_switch=False ) )
            self.add_node( h8, Node( is_switch=False ) )
            self.add_node( h9, Node( is_switch=False ) )
            self.add_node( h10, Node( is_switch=False ) )

            # Add edges
            self.add_edge( h1, s1 )
            self.add_edge( h2, s2 )
            self.add_edge( h3, s3 )
            self.add_edge( h4, s4 )
            self.add_edge( h5, s5 )
            self.add_edge( h6, s6 )
            self.add_edge( h7, s7 )
            self.add_edge( h8, s8 )
            self.add_edge( h9, s9 )
            self.add_edge( h10, s10 )

            self.add_edge( s1, s5 )
            self.add_edge( s1, s6 )
            self.add_edge( s1, s8 )
            self.add_edge( s2, s5 )
            self.add_edge( s2, s7 )
            self.add_edge( s3, s5 )
            self.add_edge( s3, s9 )
            self.add_edge( s3, s10 )
            self.add_edge( s4, s5 )



            # Consider all switches and hosts 'on'
            self.enable_all()
topos = { 'mytopo': ( lambda: MyTopo() ) }
```

A.2.2 Balanced tree matrix topology script

Listing A.3: btree.py

```
#!/usr/bin/python

from mininet.topo import Topo, Node

class MyTopo( Topo ):
    "Simple topology example."

    def __init__( self, enable_all = True ):
        "Create custom topo."

        # Add default members to class.
        super( MyTopo, self ).__init__()

        # Set Node IDs for hosts and switches
        s1=1, s2=2, s3=3, s4=4, s5=5, s6=6, s7=7
        h1=11, h2=22, h3=33, h4=44, h5=55, h6=66, h7=77
        # Add nodes
        self.add_node( s1, Node( is_switch=True ) )
       #Continue the same way till s7
        self.add_node( s7, Node( is_switch=True ) )

        self.add_node( h1, Node( is_switch=False ) )
       #Continue the same way till h7
```

```
24            self.add_node( h7, Node( is_switch=False ) )
25
26            # Add edges based on Adjacency matrix
27            self.add_edge( h1, s1 ), self.add_edge( h2, s2 )
28            self.add_edge( h3, s3 ), self.add_edge( h4, s4 )
29            self.add_edge( h5, s5 ), self.add_edge( h6, s6 )
30            self.add_edge( h7, s7 )
31
32            self.add_edge( s1, s2 ), self.add_edge( s1, s3 )
33            self.add_edge( s2, s4 ), self.add_edge( s2, s5 )
34            self.add_edge( s3, s6 ), self.add_edge( s3, s7 )
35
36            # Consider all switches and hosts 'on'
37            self.enable_all()
38 topos = { 'mytopo': ( lambda: MyTopo() ) }
```

A.2.3 Fast-gnp matrix topology script

Listing A.4: fast.py

```
1  #!/usr/bin/python
2
3  from mininet.topo import Topo, Node
4
5  class MyTopo( Topo ):
6      "Simple topology example."
7
8      def __init__( self, enable_all = True ):
9          "Create custom topo."
10
11         # Add default members to class.
12         super( MyTopo, self ).__init__()
13
14         # Set Node IDs for hosts and switches
15         s1=1
16         s2=2
17         s3=3
18         s4=4
19         s5=5
20         s6=6
21         s7=7
22         s8=8
23         s9=9
24         s10=10
25         h1=11
26         h2=22
27         h3=33
28         h4=44
29         h5=55
30         h6=66
31         h7=77
32         h8=88
33         h9=99
34         h10=100
35         # Add nodes
36          self.add_node( s1, Node( is_switch=True ) )
37         self.add_node( s2, Node( is_switch=True ) )
38         self.add_node( s3, Node( is_switch=True ) )
39         self.add_node( s4, Node( is_switch=True ) )
40         self.add_node( s5, Node( is_switch=True ) )
41         self.add_node( s6, Node( is_switch=True ) )
42         self.add_node( s7, Node( is_switch=True ) )
43         self.add_node( s8, Node( is_switch=True ) )
44         self.add_node( s9, Node( is_switch=True ) )
```

```
45            self.add_node( s10, Node( is_switch=True ) )
46
47            self.add_node( h1, Node( is_switch=False ) )
48            self.add_node( h2, Node( is_switch=False ) )
49            self.add_node( h3, Node( is_switch=False ) )
50            self.add_node( h4, Node( is_switch=False ) )
51            self.add_node( h5, Node( is_switch=False ) )
52            self.add_node( h6, Node( is_switch=False ) )
53            self.add_node( h7, Node( is_switch=False ) )
54            self.add_node( h8, Node( is_switch=False ) )
55            self.add_node( h9, Node( is_switch=False ) )
56            self.add_node( h10, Node( is_switch=False ) )
57
58            # Add edges
59            self.add_edge( h1, s1 )
60            self.add_edge( h2, s2 )
61            self.add_edge( h3, s3 )
62            self.add_edge( h4, s4 )
63            self.add_edge( h5, s5 )
64            self.add_edge( h6, s6 )
65            self.add_edge( h7, s7 )
66            self.add_edge( h8, s8 )
67            self.add_edge( h9, s9 )
68            self.add_edge( h10, s10 )
69
70            self.add_edge( s1, s3 )
71        self.add_edge( s1, s5 )
72            self.add_edge( s1, s7 )
73            self.add_edge( s1, s9 )
74            self.add_edge( s1, s10 )
75            self.add_edge( s2, s3 )
76            self.add_edge( s2, s4 )
77            self.add_edge( s2, s6 )
78            self.add_edge( s3, s8 )
79
80
81            # Consider all switches and hosts 'on'
82            self.enable_all()
83 topos = { 'mytopo': ( lambda: MyTopo() ) }
```

A.2.4 Ladder matrix topology script

Listing A.5: ladder.py

```
1  #!/usr/bin/python
2
3  from mininet.topo import Topo, Node
4
5  class MyTopo( Topo ):
6      "Simple topology example."
7
8      def __init__( self, enable_all = True ):
9          "Create custom topo."
10
11         # Add default members to class.
12         super( MyTopo, self ).__init__()
13
14         # Set Node IDs for hosts and switches
15         s1=1
16         s2=2
17         s3=3
18         s4=4
19         s5=5
20         s6=6
```

```
21            s7=7
22            s8=8
23            s9=9
24            s10=10
25            h1=11
26            h2=22
27            h3=33
28            h4=44
29            h5=55
30            h6=66
31            h7=77
32            h8=88
33            h9=99
34            h10=100
35            # Add nodes
36            self.add_node( s1, Node( is_switch=True ) )
37            self.add_node( s2, Node( is_switch=True ) )
38            self.add_node( s3, Node( is_switch=True ) )
39            self.add_node( s4, Node( is_switch=True ) )
40            self.add_node( s5, Node( is_switch=True ) )
41            self.add_node( s6, Node( is_switch=True ) )
42            self.add_node( s7, Node( is_switch=True ) )
43            self.add_node( s8, Node( is_switch=True ) )
44            self.add_node( s9, Node( is_switch=True ) )
45            self.add_node( s10, Node( is_switch=True ) )
46
47            self.add_node( h1, Node( is_switch=False ) )
48            self.add_node( h2, Node( is_switch=False ) )
49            self.add_node( h3, Node( is_switch=False ) )
50            self.add_node( h4, Node( is_switch=False ) )
51            self.add_node( h5, Node( is_switch=False ) )
52            self.add_node( h6, Node( is_switch=False ) )
53        self.add_node( h7, Node( is_switch=False ) )
54            self.add_node( h8, Node( is_switch=False ) )
55            self.add_node( h9, Node( is_switch=False ) )
56            self.add_node( h10, Node( is_switch=False ) )
57
58            # Add edges
59            self.add_edge( h1, s1 )
60            self.add_edge( h2, s2 )
61            self.add_edge( h3, s3 )
62            self.add_edge( h4, s4 )
63            self.add_edge( h5, s5 )
64            self.add_edge( h6, s6 )
65            self.add_edge( h7, s7 )
66            self.add_edge( h8, s8 )
67            self.add_edge( h9, s9 )
68            self.add_edge( h10, s10 )
69
70            self.add_edge( s1, s2 )
71            self.add_edge( s1, s6 )
72            self.add_edge( s2, s3 )
73            self.add_edge( s2, s7 )
74            self.add_edge( s3, s4 )
75            self.add_edge( s3, s8 )
76            self.add_edge( s4, s5 )
77            self.add_edge( s4, s9 )
78            self.add_edge( s5, s10 )
79
80
81            # Consider all switches and hosts 'on'
82            self.enable_all()
83    topos = { 'mytopo': ( lambda: MyTopo() ) }
```

A.2.5 Powerlaw matrix topology script

Listing A.6: power.py

```
#!/usr/bin/python

from mininet.topo import Topo, Node

class MyTopo( Topo ):
    "Simple topology example."

    def __init__( self, enable_all = True ):
        "Create custom topo."

        # Add default members to class.
        super( MyTopo, self ).__init__()

        # Set Node IDs for hosts and switches
        s1=1
        s2=2
        s3=3
        s4=4
        s5=5
        s6=6
        s7=7
        s8=8
        s9=9
        s10=10
        h1=11
        h2=22
        h3=33
        h4=44
        h5=55
        h6=66
        h7=77
        h8=88
        h9=99
        h10=100
        # Add nodes
       self.add_node( s1, Node( is_switch=True ) )
        self.add_node( s2, Node( is_switch=True ) )
        self.add_node( s3, Node( is_switch=True ) )
        self.add_node( s4, Node( is_switch=True ) )
        self.add_node( s5, Node( is_switch=True ) )
        self.add_node( s6, Node( is_switch=True ) )
        self.add_node( s7, Node( is_switch=True ) )
        self.add_node( s8, Node( is_switch=True ) )
        self.add_node( s9, Node( is_switch=True ) )
        self.add_node( s10, Node( is_switch=True ) )

        self.add_node( h1, Node( is_switch=False ) )
        self.add_node( h2, Node( is_switch=False ) )
        self.add_node( h3, Node( is_switch=False ) )
        self.add_node( h4, Node( is_switch=False ) )
        self.add_node( h5, Node( is_switch=False ) )
        self.add_node( h6, Node( is_switch=False ) )
        self.add_node( h7, Node( is_switch=False ) )
        self.add_node( h8, Node( is_switch=False ) )
        self.add_node( h9, Node( is_switch=False ) )
        self.add_node( h10, Node( is_switch=False ) )

        # Add edges
        self.add_edge( h1, s1 )
        self.add_edge( h2, s2 )
        self.add_edge( h3, s3 )
        self.add_edge( h4, s4 )
        self.add_edge( h5, s5 )
        self.add_edge( h6, s6 )
```

```
            self.add_edge( h7, s7 )
            self.add_edge( h8, s8 )
            self.add_edge( h9, s9 )
            self.add_edge( h10, s10 )

            self.add_edge( s1, s5 )
        self.add_edge( s1, s6 )
            self.add_edge( s1, s9 )
            self.add_edge( s2, s7 )
            self.add_edge( s2, s9 )
            self.add_edge( s2, s10 )
            self.add_edge( s3, s8 )
            self.add_edge( s3, s10 )
            self.add_edge( s4, s5 )


            # Consider all switches and hosts 'on'
            self.enable_all()
topos = { 'mytopo': ( lambda: MyTopo() ) }
```

A.2.6 Random Regular matrix topology script

Listing A.7: regular.py

```
#!/usr/bin/python

from mininet.topo import Topo, Node

class MyTopo( Topo ):
    "Simple topology example."

    def __init__( self, enable_all = True ):
        "Create custom topo."

        # Add default members to class.
        super( MyTopo, self ).__init__()

        # Set Node IDs for hosts and switches
        s1=1
        s2=2
        s3=3
        s4=4
        s5=5
        s6=6
        s7=7
        s8=8
        s9=9
        s10=10
        h1=11
        h2=22
        h3=33
        h4=44
        h5=55
        h6=66
        h7=77
        h8=88
        h9=99
        h10=100
        # Add nodes
    self.add_node( s1, Node( is_switch=True ) )
        self.add_node( s2, Node( is_switch=True ) )
        self.add_node( s3, Node( is_switch=True ) )
        self.add_node( s4, Node( is_switch=True ) )
        self.add_node( s5, Node( is_switch=True ) )
```

```
41            self.add_node( s6, Node( is_switch=True ) )
42            self.add_node( s7, Node( is_switch=True ) )
43            self.add_node( s8, Node( is_switch=True ) )
44            self.add_node( s9, Node( is_switch=True ) )
45            self.add_node( s10, Node( is_switch=True ) )
46
47            self.add_node( h1, Node( is_switch=False ) )
48            self.add_node( h2, Node( is_switch=False ) )
49            self.add_node( h3, Node( is_switch=False ) )
50            self.add_node( h4, Node( is_switch=False ) )
51            self.add_node( h5, Node( is_switch=False ) )
52            self.add_node( h6, Node( is_switch=False ) )
53            self.add_node( h7, Node( is_switch=False ) )
54            self.add_node( h8, Node( is_switch=False ) )
55            self.add_node( h9, Node( is_switch=False ) )
56            self.add_node( h10, Node( is_switch=False ) )
57
58            # Add edges
59            self.add_edge( h1, s1 )
60            self.add_edge( h2, s2 )
61            self.add_edge( h3, s3 )
62            self.add_edge( h4, s4 )
63            self.add_edge( h5, s5 )
64            self.add_edge( h6, s6 )
65            self.add_edge( h7, s7 )
66            self.add_edge( h8, s8 )
67            self.add_edge( h9, s9 )
68            self.add_edge( h10, s10 )
69
70            self.add_edge( s1, s2 )
71      self.add_edge( s1, s5 )
72            self.add_edge( s1, s6 )
73            self.add_edge( s1, s7 )
74            self.add_edge( s2, s10 )
75            self.add_edge( s3, s4 )
76            self.add_edge( s3, s5 )
77            self.add_edge( s3, s8 )
78            self.add_edge( s4, s9 )
79
80
81            # Consider all switches and hosts 'on'
82            self.enable_all()
83    topos = { 'mytopo': ( lambda: MyTopo() ) }
```

A.2.7 Star matrix topology script

Listing A.8: star.py

```
1   #!/usr/bin/python
2
3   from mininet.topo import Topo, Node
4
5   class MyTopo( Topo ):
6       "Simple topology example."
7
8       def __init__( self, enable_all = True ):
9           "Create custom topo."
10
11          # Add default members to class.
12          super( MyTopo, self ).__init__()
13
14          # Set Node IDs for hosts and switches
15          s1=1
16          s2=2
```

```
            s3=3
            s4=4
            s5=5
            s6=6
            s7=7
            s8=8
            s9=9
            s10=10
            h1=11
            h2=22
            h3=33
            h4=44
            h5=55
            h6=66
            h7=77
            h8=88
            h9=99
            h10=100
            # Add nodes
 self.add_node( s1, Node( is_switch=True ) )
            self.add_node( s2, Node( is_switch=True ) )
            self.add_node( s3, Node( is_switch=True ) )
            self.add_node( s4, Node( is_switch=True ) )
            self.add_node( s5, Node( is_switch=True ) )
            self.add_node( s6, Node( is_switch=True ) )
            self.add_node( s7, Node( is_switch=True ) )
            self.add_node( s8, Node( is_switch=True ) )
            self.add_node( s9, Node( is_switch=True ) )
            self.add_node( s10, Node( is_switch=True ) )

            self.add_node( h1, Node( is_switch=False ) )
            self.add_node( h2, Node( is_switch=False ) )
            self.add_node( h3, Node( is_switch=False ) )
            self.add_node( h4, Node( is_switch=False ) )
            self.add_node( h5, Node( is_switch=False ) )
            self.add_node( h6, Node( is_switch=False ) )
            self.add_node( h7, Node( is_switch=False ) )
            self.add_node( h8, Node( is_switch=False ) )
            self.add_node( h9, Node( is_switch=False ) )
            self.add_node( h10, Node( is_switch=False ) )

            # Add edges
            self.add_edge( h1, s1 )
            self.add_edge( h2, s2 )
            self.add_edge( h3, s3 )
            self.add_edge( h4, s4 )
            self.add_edge( h5, s5 )
            self.add_edge( h6, s6 )
            self.add_edge( h7, s7 )
            self.add_edge( h8, s8 )
            self.add_edge( h9, s9 )
            self.add_edge( h10, s10 )

            self.add_edge( s1, s2 )
       self.add_edge( s1, s3 )
            self.add_edge( s1, s4 )
            self.add_edge( s1, s5 )
            self.add_edge( s1, s6 )
            self.add_edge( s1, s7 )
            self.add_edge( s1, s8 )
            self.add_edge( s1, s9 )
            self.add_edge( s1, s10 )

            # Consider all switches and hosts 'on'
            self.enable_all()
topos = { 'mytopo': ( lambda: MyTopo() ) }
```

A.2.8 Watts matrix topology script

Listing A.9: watts.py

```
#!/usr/bin/python

from mininet.topo import Topo, Node

class MyTopo( Topo ):
    "Simple topology example."

    def __init__( self, enable_all = True ):
        "Create custom topo."

        # Add default members to class.
        super( MyTopo, self ).__init__()

        # Set Node IDs for hosts and switches
        s1=1
        s2=2
        s3=3
        s4=4
        s5=5
        s6=6
        s7=7
        s8=8
        s9=9
        s10=10
        h1=11
        h2=22
        h3=33
        h4=44
        h5=55
        h6=66
        h7=77
        h8=88
        h9=99
        h10=100
        # Add nodes
 self.add_node( s1, Node( is_switch=True ) )
        self.add_node( s2, Node( is_switch=True ) )
        self.add_node( s3, Node( is_switch=True ) )
        self.add_node( s4, Node( is_switch=True ) )
        self.add_node( s5, Node( is_switch=True ) )
        self.add_node( s6, Node( is_switch=True ) )
        self.add_node( s7, Node( is_switch=True ) )
        self.add_node( s8, Node( is_switch=True ) )
        self.add_node( s9, Node( is_switch=True ) )
        self.add_node( s10, Node( is_switch=True ) )

        self.add_node( h1, Node( is_switch=False ) )
        self.add_node( h2, Node( is_switch=False ) )
        self.add_node( h3, Node( is_switch=False ) )
        self.add_node( h4, Node( is_switch=False ) )
        self.add_node( h5, Node( is_switch=False ) )
        self.add_node( h6, Node( is_switch=False ) )
        self.add_node( h7, Node( is_switch=False ) )
        self.add_node( h8, Node( is_switch=False ) )
        self.add_node( h9, Node( is_switch=False ) )
        self.add_node( h10, Node( is_switch=False ) )

        # Add edges
        self.add_edge( h1, s1 )
        self.add_edge( h2, s2 )
        self.add_edge( h3, s3 )
        self.add_edge( h4, s4 )
        self.add_edge( h5, s5 )
        self.add_edge( h6, s6 )
```

```
            self.add_edge( h7, s7 )
            self.add_edge( h8, s8 )
            self.add_edge( h9, s9 )
            self.add_edge( h10, s10 )

            self.add_edge( s1, s2 )
        self.add_edge( s1, s7 )
            self.add_edge( s1, s8 )
            self.add_edge( s1, s10 )
            self.add_edge( s2, s3 )
            self.add_edge( s2, s5 )
            self.add_edge( s3, s6 )
            self.add_edge( s4, s10 )
            self.add_edge( s4, s9 )
            # Consider all switches and hosts 'on'
            self.enable_all()
topos = { 'mytopo': ( lambda: MyTopo() ) }
```

A.3 Mathematica main commands

```
a = Import["I:\\files\\btree\\bt2.txt", {"Data"}]

g = AdjacencyGraph[a]
SmoothHistogram[a]

N[Mean[VertexDegree[g]]]
d = DiagonalMatrix[VertexDegree[g]]
l = d − a

e1 = N[Eigenvalues[a]]
Sort[e1]
ne1 = Normalize[e1]
SmoothHistogram[ne1]
SmoothHistogram[e1]

e2 = N[Eigenvalues[l]]
Sort[e2]
ne2 = Normalize[e2]
SmoothHistogram[ne2]
```

Appendix B

Results

B.1 Balanced Tree 21x21 Matrix results

```
a = {{0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, {1,
   0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, {1, 0,
   0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, {1, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0}, {1, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, {0, 1, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, {0, 1, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, {0, 1, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, {0, 1, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 1, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0}, {0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0}, {0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0}, {0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0}, {0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0}}

   d = {{4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, {0,
   5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, {0, 0,
   5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0,
   5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0,
   5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0,
   1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0,
   1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0,
   1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0,
   1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0, 0,
   1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   1, 0, 0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   1, 0, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   1, 0, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   1, 0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   1, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   1, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   1, 0, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   1, 0, 0}, {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

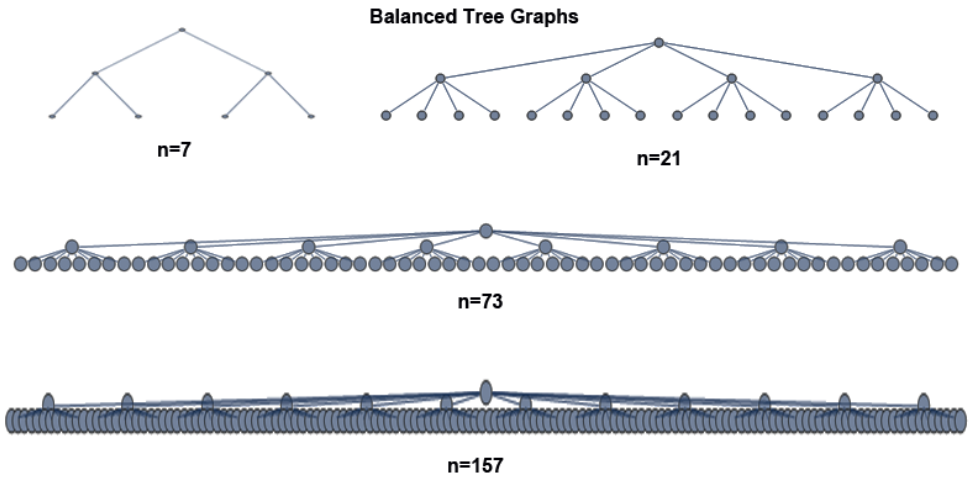

Figure B.1: Balanced tree graphs

```
   1, 0}, {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   1}}

   l = {{4, −1, −1, −1, −1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0}, {−1, 5, 0, 0, 0, −1, −1, −1, −1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0}, {−1, 0, 5, 0, 0, 0, 0, 0, 0, −1, −1, −1, −1, 0, 0, 0, 0, 0,
   0, 0, 0}, {−1, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, −1, −1, −1, −1,
   0, 0, 0, 0}, {−1, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, −1, −1, −1, −1}, {0, −1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0, 0}, {0, −1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 0, 0, 0}, {0, −1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0, 0}, {0, −1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 0, 0}, {0, 0, −1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
   0, 0, 0, 0}, {0, 0, −1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
     0, 0, 0}, {0, 0, −1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
   0, 0, 0}, {0, 0, −1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
     0, 0}, {0, 0, 0, −1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
   0, 0}, {0, 0, 0, −1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
     0}, {0, 0, 0, −1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
   0}, {0, 0, 0, 0, −1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
   0}, {0, 0, 0, 0, −1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
   0}, {0, 0, 0, 0, −1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
   0}, {0, 0, 0, 0, −1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   1}}

   e1 = {−2.82843, 2.82843, −2., −2., −2., 2., 2., 2., 0., 0., 0., 0., 0., \
0., 0., 0., 0., 0., 0., 0., 0.}

e2 = {7., 5.82843, 5.82843, 5.82843, 3., 1., 1., 1., 1., 1., 1., 1., \
1., 1., 1., 1., 0.171573, 0.171573, 0.171573, 0.}
```

B.2 Balanced tree full graphs and histograms

```
avg−degree = 4.8
min−degree = 3.
```

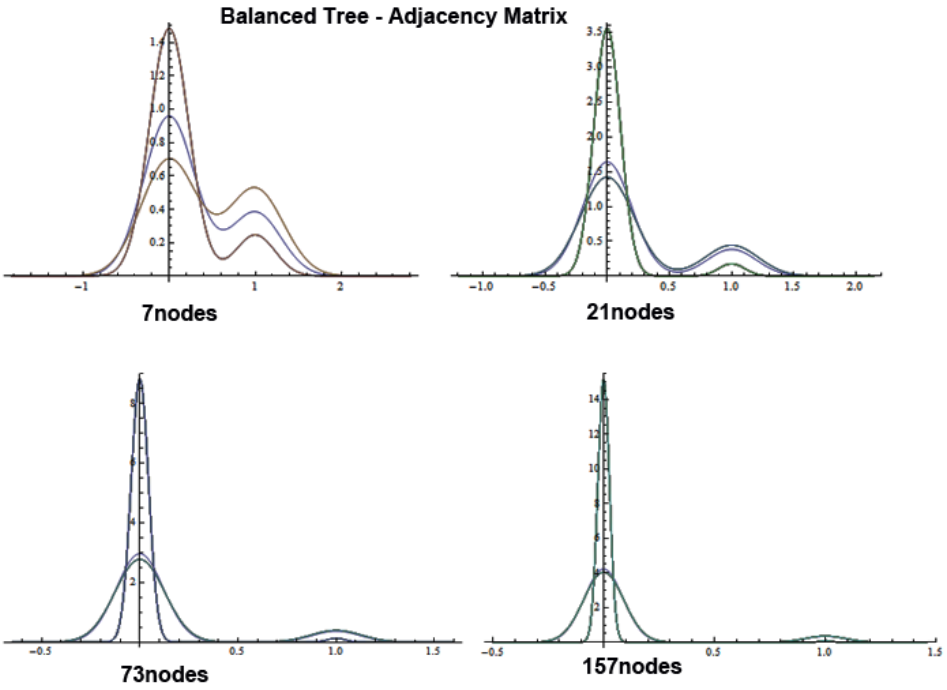

Figure B.2: Balanced tree adjacency matrix histograms

B.3 Barabasi results

B.3.1 Barabasi 10 nodes results

```
sort−eigen =
    {−2.57236,−2.05252,−1.73435,−1.15509,−0.686725,−0.310336,0.26712,1.39247,1.58523,5.26657}
sort−lapl−eigen = {0.,2.34773,2.47428,3.,4.29937,5.36935,5.94509,6.71205,7.85212,10.}
betweenness = {0.,0.,0.25,0.75,10.75,2.,3.08333,2.66667,1.16667,0.333333}
```

B.3.2 Barabasi 20 nodes results

```
avg−degree = 6.4
min−degree = 3.
sort−eigen =
    {−2.57236,−2.05252,−1.73435,−1.15509,−0.686725,−0.310336,0.26712,1.39247,1.58523,5.26657}
sort−lapl−eigen = {0.,2.34773,2.47428,3.,4.29937,5.36935,5.94509,6.71205,7.85212,10.}
betweenness =
    {4.2,1.24286,17.1937,0.642857,49.9944,9.08016,2.78333,10.3984,4.79286,0.416667,2.,4.97024,
    7.86905,4.20675,0.366667,0.2,6.93135,4.54405,1.58333,0.583333}
```

B.3.3 Barabasi 40 nodes results

```
avg−degree = 7.2

min−degree = 4.

sort−eigen = {−4.65743, −4.16902, −3.98983, −3.8439, −3.20181, −2.76676, −2.57371, \
−2.23669, −2.11446, −1.94703, −1.7839, −1.69394, −1.54367, −1.38193, \
−1.28256, −0.799529, −0.478849, −0.395272, −0.302994, −0.0990431, \
0.0338861, 0.0636623, 0.233581, 0.463215, 0.656049, 0.76768, \
0.948211, 1.02902, 1.32358, 1.3894, 1.74361, 2.04303, 2.08626, \
2.71083, 2.78773, 2.95423, 3.0884, 3.48194, 4.20388, 9.25414}

sort−lapl−eigen = {0., 2.06843, 2.60492, 2.61906, 2.79888, 2.98762, 3.10315, 3.28276, \
3.35677, 3.57012, 3.6036, 3.65672, 3.86555, 3.91197, 4.3746, 4.62442, \
5.10328, 5.1729, 5.22275, 5.50292, 5.80553, 5.84698, 6.16264, \
6.70411, 6.94016, 7.03965, 8.06023, 8.24835, 8.41691, 9.78437, \
10.1429, 10.9018, 11.7755, 12.2471, 12.9016, 13.5137, 14.7226, \
15.4912, 17.7074, 24.1569}

betweenness = {80.7225, 48.2999, 11.2487, 8.26853, 165.313, 33.7181, 13.1544, \
58.8176, 36.0457, 45.8687, 16.773, 48.9265, 11.562, 11.1491, 41.1323, \
16.5977, 34.5569, 4.29232, 10.5631, 8.47143, 0.30303, 2.83492, \
2.1856, 9.79906, 0.538095, 22.3, 1.58374, 1.03452, 8.91231, 11.1505, \
1.64156, 4.47976, 4.26424, 1.47357, 2.81329, 1.95833, 4.51776, \
1.90321, 1.33929, 1.48611}
```

B.3.4 Barabasi 80 nodes results

```
avg−degree = 7.6

min−degree = 4.

sort−eigen = {−6.14274, −5.26783, −5.15604, −4.90936, −4.43508, −3.84707, \
−3.55874, −3.29773, −3.20141, −3.14436, −3.05154, −3.00798, −2.81673, \
−2.71371, −2.57907, −2.43141, −2.33836, −2.22114, −2.13909, −1.81985, \
−1.63937, −1.60873, −1.51646, −1.4429, −1.3117, −1.23271, −1.13655, \
−1.01375, −0.93079, −0.798686, −0.716545, −0.63263, −0.571356, \
−0.518345, −0.486365, −0.37278, −0.274, −0.199978, −0.0918715, \
−0.0620031, −0.0296272, 0.119178, 0.141343, 0.198378, 0.287235, \
```

```
0.323416, 0.486081, 0.569342, 0.603008, 0.738431, 0.783528, 0.843308, \
0.902242, 1.04694, 1.12616, 1.20989, 1.32, 1.42299, 1.51248, 1.69873, \
1.76999, 1.90043, 2.018, 2.21445, 2.24058, 2.32826, 2.5063, 2.60177, \
2.65444, 2.90455, 3.05889, 3.10719, 3.32247, 3.73306, 3.84471, \
4.07338, 4.3612, 4.56033, 5.44168, 10.692}

sort−lapl−eigen = {0., 2.17143, 2.36393, 2.40577, 2.50342, 2.60857, 2.69635, 2.89551, \
2.99762, 3.03558, 3.14602, 3.19273, 3.25833, 3.29645, 3.35006, \
3.42195, 3.46638, 3.52918, 3.55616, 3.59436, 3.64534, 3.71251, \
3.76965, 3.78861, 3.81984, 3.89328, 3.91648, 3.94053, 3.94487, \
4.10011, 4.28232, 4.34311, 4.44414, 4.60795, 4.92665, 5.0639, \
5.24961, 5.31928, 5.33363, 5.4117, 5.59894, 5.69677, 5.79615, \
5.94639, 6.14527, 6.23977, 6.49698, 6.53773, 6.62627, 6.73214, \
7.15532, 7.30279, 7.34945, 7.82782, 8.1003, 8.39354, 8.67488, \
8.82792, 9.13617, 9.48898, 9.74491, 10.2874, 10.4596, 10.6387, \
10.9943, 11.4872, 11.7852, 12.7016, 13.4845, 13.8165, 14.0011, \
14.7219, 15.8879, 17.4746, 18.1626, 20.5453, 22.497, 25.0359, \
25.8743, 29.3228}

betweenness = {34.0116, 73.6317, 5.80631, 96.2738, 347.325, 350.089, 157.565, \
252.661, 476.293, 149.641, 237.031, 70.8148, 117.83, 138.314, \
10.2163, 72.0899, 48.2935, 106.995, 120.873, 64.4559, 32.1186, \
42.5072, 10.6842, 51.7596, 40.4173, 51.2295, 39.7171, 33.7087, \
17.349, 31.189, 44.4402, 20.2877, 27.9239, 12.6177, 50.2636, 11.5661, \
57.3251, 90.5329, 4.65535, 9.59626, 23.6626, 17.8744, 6.98936, \
0.303091, 13.8156, 6.44642, 18.5444, 27.0709, 6.04439, 5.95491, \
10.0147, 5.02571, 28.6068, 11.069, 13.6856, 30.2345, 21.2052, \
13.3435, 0.784098, 8.53215, 6.80227, 6.42549, 20.7477, 14.0606, \
4.94879, 16.6388, 3.6531, 0.301786, 5.53204, 7.05324, 13.2743, \
36.1808, 16.2045, 3.52058, 9.46456, 5.28058, 3.60173, 2.64506, \
2.71055, 3.64609}
```

B.3.5 Barabasi 160 nodes results

```
avg−degree = 7.6

min−degree = 4.

sort−eigen = {−6.14274, −5.26783, −5.15604, −4.90936, −4.43508, −3.84707, \
−3.55874, −3.29773, −3.20141, −3.14436, −3.05154, −3.00798, −2.81673, \
−2.71371, −2.57907, −2.43141, −2.33836, −2.22114, −2.13909, −1.81985, \
−1.63937, −1.60873, −1.51646, −1.4429, −1.3117, −1.23271, −1.13655, \
−1.01375, −0.93079, −0.798686, −0.716545, −0.63263, −0.571356, \
−0.518345, −0.486365, −0.37278, −0.274, −0.199978, −0.0918715, \
−0.0620031, −0.0296272, 0.119178, 0.141343, 0.198378, 0.287235, \
0.323416, 0.486081, 0.569342, 0.603008, 0.738431, 0.783528, 0.843308, \
0.902242, 1.04694, 1.12616, 1.20989, 1.32, 1.42299, 1.51248, 1.69873, \
1.76999, 1.90043, 2.018, 2.21445, 2.24058, 2.32826, 2.5063, 2.60177, \
2.65444, 2.90455, 3.05889, 3.10719, 3.32247, 3.73306, 3.84471, \
4.07338, 4.3612, 4.56033, 5.44168, 10.692}

sort−lapl−eigen = {0., 2.17143, 2.36393, 2.40577, 2.50342, 2.60857, 2.69635, 2.89551, \
2.99762, 3.03558, 3.14602, 3.19273, 3.25833, 3.29645, 3.35006, \
3.42195, 3.46638, 3.52918, 3.55616, 3.59436, 3.64534, 3.71251, \
3.76965, 3.78861, 3.81984, 3.89328, 3.91648, 3.94053, 3.94487, \
4.10011, 4.28232, 4.34311, 4.44414, 4.60795, 4.92665, 5.0639, \
5.24961, 5.31928, 5.33363, 5.4117, 5.59894, 5.69677, 5.79615, \
5.94639, 6.14527, 6.23977, 6.49698, 6.53773, 6.62627, 6.73214, \
7.15532, 7.30279, 7.34945, 7.82782, 8.1003, 8.39354, 8.67488, \
8.82792, 9.13617, 9.48898, 9.74491, 10.2874, 10.4596, 10.6387, \
10.9943, 11.4872, 11.7852, 12.7016, 13.4845, 13.8165, 14.0011, \
14.7219, 15.8879, 17.4746, 18.1626, 20.5453, 22.497, 25.0359, \
25.8743, 29.3228}

```

```
betweenness = {34.0116, 73.6317, 5.80631, 96.2738, 347.325, 350.089, 157.565, \
252.661, 476.293, 149.641, 237.031, 70.8148, 117.83, 138.314, \
10.2163, 72.0899, 48.2935, 106.995, 120.873, 64.4559, 32.1186, \
42.5072, 10.6842, 51.7596, 40.4173, 51.2295, 39.7171, 33.7087, \
17.349, 31.189, 44.4402, 20.2877, 27.9239, 12.6177, 50.2636, 11.5661, \
57.3251, 90.5329, 4.65535, 9.59626, 23.6626, 17.8744, 6.98936, \
0.303091, 13.8156, 6.44642, 18.5444, 27.0709, 6.04439, 5.95491, \
10.0147, 5.02571, 28.6068, 11.069, 13.6856, 30.2345, 21.2052, \
13.3435, 0.784098, 8.53215, 6.80227, 6.42549, 20.7477, 14.0606, \
4.94879, 16.6388, 3.6531, 0.301786, 5.53204, 7.05324, 13.2743, \
36.1808, 16.2045, 3.52058, 9.46456, 5.28058, 3.60173, 2.64506, \
2.71055, 3.64609}
```

B.4 Fast-gnp results

B.4.1 Fast-gnp 10 nodes results

```
avg−degree = 4.2

min−degree = 2.

sort−eigen = {−2.86599, −2.28315, −1.63347, −0.910937, −0.337529, 0.0488796, \
0.804017, 1.24168, 1.3636, 4.5729}

sort−lapl−eigen = {0., 1.51309, 2.05278, 3.13456, 3.97264, 4.41375, 5.36733, 6.50421, \
7.22963, 7.81202}

betweenness = {3.41667, 4.75, 4.75, 1.16667, 2.66667, 0.916667, 0.333333, 1.66667, \
6.33333, 0.}
```

B.4.2 Fast-gnp 20 nodes results

```
avg−degree = 7.8

min−degree = 4.

sort−eigen = {−3.71667, −3.26116, −3.14537, −2.57166, −2.36302, −2.22384, \
−1.88862, −1.17638, −0.760219, −0.667429, 0.12277, 0.388541, \
0.472681, 1.17462, 1.46284, 1.97595, 2.15014, 2.52794, 3.14159, \
8.35729}

sort−lapl−eigen = {0., 3.26784, 3.911, 4.40042, 4.54819, 5.67236, 5.92009, 6.50154, \
6.94297, 7.62839, 8.15614, 8.66681, 9.11614, 9.77654, 10.1508, \
10.9854, 11.6724, 12.3693, 12.6105, 13.7033}

sbetweenness = {2.61111, 0.597222, 3.80675, 3.74167, 8.61389, 3.75119, 4.81786, 5.2, \
1.63889, 10.5639, 4.25556, 8.89444, 17.0639, 4.61667, 5.83452, \
14.2917, 1.59286, 7.20397, 2.01944, 2.88452}
```

B.4.3 Fast-gnp 40 nodes results

```
avg−degree = 16.6

```

```
min−degree = 9.

sort−eigen = {−5.74595, −5.46519, −5.21035, −4.62369, −4.44463, −4.1438, −3.95815, \
 −3.69551, −3.29719, −2.88841, −2.74622, −2.50811, −2.41213, −2.12114, \
 −1.79051, −1.45317, −1.09543, −0.987074, −0.747341, −0.428114, \
0.00336968, 0.192183, 0.331997, 0.736578, 0.918763, 1.25024, 1.60496, \
1.73424, 1.99723, 2.06922, 2.35868, 2.65569, 2.84331, 3.08367, 3.475, \
3.74583, 3.9806, 4.26515, 5.17633, 17.3391}

sort−lapl−eigen = {0., 7.94433, 9.34508, 10.2428, 10.6491, 10.9355, 11.5091, 11.8315, \
12.2091, 12.8038, 13.286, 13.85, 14.1311, 14.4194, 15.0133, 15.153, \
15.5298, 15.8504, 16.3794, 16.5027, 16.6564, 16.9453, 17.0986, \
17.6894, 18.1782, 18.4033, 19.1172, 19.4472, 20.2024, 20.6999, \
21.4164, 21.5641, 21.9339, 22.532, 22.6424, 23.5809, 23.7152, \
24.2817, 24.8452, 25.4647}

betweenness = {11.4618, 11.0808, 5.2732, 8.87507, 4.11468, 7.41527, 20.2098, \
7.42734, 13.5913, 11.9903, 12.0767, 11.8601, 7.91785, 13.062, \
9.37572, 10.9708, 3.51908, 8.28168, 18.9025, 7.56595, 10.7037, \
12.1977, 5.75156, 11.5206, 9.08224, 9.0684, 16.7719, 15.1819, \
22.1848, 16.8954, 18.1012, 13.4211, 7.85729, 6.44567, 13.4887, \
6.07904, 20.1327, 10.4449, 11.027, 7.67218}
```

B.4.4 Fast-gnp 80 nodes results

```
avg−degree = 16.6

min−degree = 9.

sort−eigen = {−5.74595, −5.46519, −5.21035, −4.62369, −4.44463, −4.1438, −3.95815, \
 −3.69551, −3.29719, −2.88841, −2.74622, −2.50811, −2.41213, −2.12114, \
 −1.79051, −1.45317, −1.09543, −0.987074, −0.747341, −0.428114, \
0.00336968, 0.192183, 0.331997, 0.736578, 0.918763, 1.25024, 1.60496, \
1.73424, 1.99723, 2.06922, 2.35868, 2.65569, 2.84331, 3.08367, 3.475, \
3.74583, 3.9806, 4.26515, 5.17633, 17.3391}

sort−lapl−eigen = {0., 7.94433, 9.34508, 10.2428, 10.6491, 10.9355, 11.5091, 11.8315, \
12.2091, 12.8038, 13.286, 13.85, 14.1311, 14.4194, 15.0133, 15.153, \
15.5298, 15.8504, 16.3794, 16.5027, 16.6564, 16.9453, 17.0986, \
17.6894, 18.1782, 18.4033, 19.1172, 19.4472, 20.2024, 20.6999, \
21.4164, 21.5641, 21.9339, 22.532, 22.6424, 23.5809, 23.7152, \
24.2817, 24.8452, 25.4647}

betweenness = {11.4618, 11.0808, 5.2732, 8.87507, 4.11468, 7.41527, 20.2098, \
7.42734, 13.5913, 11.9903, 12.0767, 11.8601, 7.91785, 13.062, \
9.37572, 10.9708, 3.51908, 8.28168, 18.9025, 7.56595, 10.7037, \
12.1977, 5.75156, 11.5206, 9.08224, 9.0684, 16.7719, 15.1819, \
22.1848, 16.8954, 18.1012, 13.4211, 7.85729, 6.44567, 13.4887, \
6.07904, 20.1327, 10.4449, 11.027, 7.67218}
```

B.4.5 Fast-gnp 160 nodes results

```
avg−degree = 64.425

min−degree = 50.

sort−eigen = {−12.4856, −11.8568, −11.6868, −11.5112, −11.1976, −10.8567, \
 −10.7273, −10.4976, −10.2555, −10.0945, −9.89372, −9.78111, −9.64704, \
 −9.30866, −9.14928, −8.96283, −8.85802, −8.78943, −8.54149, −8.30547, \
 −8.01329, −7.96344, −7.8156, −7.63104, −7.43512, −7.33514, −7.07392, \
```

```
−7.05979, −6.97944, −6.77319, −6.67394, −6.63545, −6.40065, −6.17611, \
−6.05316, −5.87323, −5.77577, −5.5582, −5.42493, −5.26435, −5.21374, \
−5.08515, −4.95381, −4.8343, −4.77508, −4.6277, −4.51544, −4.48243, \
−4.19959, −4.12734, −4.06039, −3.87885, −3.85561, −3.59435, −3.56403, \
−3.38168, −3.3166, −3.25314, −3.01136, −2.87289, −2.65549, −2.56983, \
−2.50592, −2.39033, −2.3391, −2.05359, −1.98279, −1.86878, −1.77617, \
−1.72314, −1.46267, −1.26661, −1.20821, −0.983293, −0.817585, \
−0.774002, −0.680638, −0.565946, −0.44012, −0.397806, −0.304719, \
−0.128209, −0.0577045, 0.00670477, 0.164161, 0.298345, 0.48089, \
0.534409, 0.648735, 0.755341, 0.910123, 0.997627, 1.19976, 1.27934, \
1.34864, 1.49013, 1.5616, 1.71432, 1.76704, 1.94092, 1.99022, \
2.18712, 2.25673, 2.30372, 2.68656, 2.7459, 2.94365, 3.00807, \
3.19605, 3.33068, 3.35507, 3.53434, 3.72521, 3.92529, 4.05303, \
4.14669, 4.34061, 4.58361, 4.60401, 4.71472, 4.95346, 4.98542, \
5.00161, 5.30585, 5.36987, 5.48197, 5.60983, 5.68022, 6.01226, \
6.13423, 6.27899, 6.35942, 6.5179, 6.56495, 6.86879, 7.02172, \
7.34093, 7.44564, 7.55786, 7.59466, 7.66128, 7.84827, 8.03219, \
8.11712, 8.26483, 8.41449, 8.61836, 8.85101, 8.8875, 9.03365, \
9.23638, 9.45695, 9.59585, 9.76186, 9.99043, 10.3872, 10.5529, \
10.7107, 11.5424, 65.1002}

sort−lapl−eigen = {0., 46.8829, 47.5697, 48.1398, 48.8975, 49.1709, 49.2424, 49.8157, \
50.0417, 50.5792, 50.9272, 51.2575, 51.5101, 51.7665, 51.8444, \
52.2213, 52.5333, 52.7501, 52.875, 53.198, 53.4757, 53.5377, 53.706, \
54.0293, 54.2292, 54.3818, 54.8299, 54.9981, 55.1317, 55.3882, \
55.5083, 55.6931, 55.9463, 56.198, 56.3229, 56.6905, 56.8017, \
56.9979, 57.1391, 57.4923, 57.6561, 57.8365, 58.0629, 58.1271, \
58.3835, 58.462, 58.6994, 58.9728, 59.2414, 59.3836, 59.4859, \
59.5461, 59.6861, 59.8365, 60.2173, 60.3393, 60.5118, 60.5474, \
60.8505, 60.8804, 60.9199, 61.3779, 61.5023, 61.6027, 61.8124, \
62.0651, 62.2919, 62.4505, 62.8089, 62.913, 63.0336, 63.2458, \
63.4748, 63.5463, 63.6728, 63.912, 64.0021, 64.1689, 64.2182, \
64.5369, 64.749, 64.8317, 65.0022, 65.1871, 65.4072, 65.4932, \
65.6454, 65.9664, 66.0621, 66.2161, 66.3165, 66.4815, 66.8258, \
66.9274, 67.0072, 67.1429, 67.4063, 67.5868, 67.6593, 67.975, \
68.0985, 68.426, 68.5709, 68.7502, 69.0238, 69.204, 69.295, 69.3665, \
69.5039, 69.7103, 69.936, 70.0167, 70.2842, 70.7355, 70.8661, \
70.9643, 71.1184, 71.1712, 71.5009, 71.8057, 72.0423, 72.3275, \
72.3732, 72.5288, 72.8626, 73.0067, 73.0448, 73.3143, 73.4267, \
73.6989, 73.7715, 73.8996, 74.3111, 74.5014, 74.6879, 75.0381, \
75.2207, 75.4889, 75.576, 76.0097, 76.3339, 76.5929, 76.7583, 76.957, \
77.1915, 77.7768, 77.9918, 78.1741, 78.3713, 78.7278, 79.3904, \
79.5979, 79.9737, 80.5399, 81.3847, 81.7457, 82.3233, 82.8217, \
84.6684, 85.3086}

betweenness = {52.0739, 46.5943, 31.8146, 46.0745, 54.9289, 38.113, 46.873, \
37.0337, 35.6276, 56.5012, 37.041, 41.8487, 46.3067, 61.1723, \
45.1272, 28.0512, 48.3727, 27.5322, 50.4675, 41.9321, 43.8282, \
59.0161, 40.5139, 36.6789, 33.3819, 42.9707, 38.3807, 54.8581, \
51.2898, 66.0091, 46.9604, 65.8122, 70.7576, 55.4983, 53.8664, \
39.6275, 60.5745, 44.1738, 30.7764, 44.4845, 36.6784, 46.0774, \
29.5851, 35.5156, 51.3145, 52.5915, 39.9206, 49.2113, 44.4667, \
46.343, 48.0918, 51.8431, 70.246, 38.2251, 36.1815, 55.6802, 56.1437, \
36.121, 53.8166, 58.217, 60.5993, 50.4505, 59.4212, 40.4407, 29.5383, \
41.8746, 47.6505, 54.9537, 42.9059, 56.6057, 30.1979, 49.638, \
48.5417, 35.716, 48.2082, 43.5248, 37.5947, 45.6204, 48.744, 39.7411, \
46.7909, 45.629, 31.6847, 47.7563, 43.2608, 66.1437, 37.4507, \
74.9062, 53.3285, 51.1317, 51.6964, 76.2276, 50.8649, 47.5007, \
32.9509, 52.0413, 61.2736, 49.8524, 59.2792, 41.5922, 33.901, \
41.3251, 40.2356, 33.2239, 51.8884, 48.1593, 54.3746, 33.8358, \
61.9807, 47.248, 56.592, 61.1162, 47.2401, 29.2616, 34.9814, 31.5377, \
59.1455, 45.4314, 45.8365, 40.7668, 39.8894, 33.6893, 53.8682, \
47.7365, 39.892, 47.451, 47.9378, 32.8896, 47.3555, 68.5698, 49.693, \
42.3647, 43.9827, 62.5245, 69.5175, 46.8046, 63.8194, 29.604, \
40.6831, 59.5783, 42.2093, 37.6081, 46.3355, 55.178, 50.1098, \
65.1668, 50.35, 46.8766, 69.3418, 43.092, 37.4109, 39.4729, 57.6051, \
39.0328, 49.5311, 53.6923, 61.9945, 33.2626, 49.9174, 49.3357}
```

B.5 Ladder results

B.5.1 Ladder 5 nodes results

```
avg−degree = 2.6

min−degree = 2.

sort−eigen = {−2.73205, −2., −1., −0.732051, 0., 0., 0.732051, 1., 2., 2.73205}

sort−lapl−eigen = {0., 0.381966, 1.38197, 2., 2.38197, 2.61803, 3.38197, 3.61803, \
4.61803, 5.61803}

betweenness = {1.28333, 8.36667, 10.7, 8.36667, 1.28333, 1.28333, 8.36667, 10.7, \
8.36667, 1.28333}
```

B.5.2 Ladder 10 nodes results

```
avg−degree = 2.8

min−degree = 2.

sort−eigen = {−2.91899, −2.68251, −2.30972, −1.83083, −1.28463, −0.918986, \
 −0.71537, −0.682507, −0.309721, −0.16917, 0.16917, 0.309721, \
0.682507, 0.71537, 0.918986, 1.28463, 1.83083, 2.30972, 2.68251, \
2.91899}

sort−lapl−eigen = {0., 0.097887, 0.381966, 0.824429, 1.38197, 2., 2., 2.09789, 2.38197, \
2.61803, 2.82443, 3.17557, 3.38197, 3.61803, 3.90211, 4., 4.61803, \
5.17557, 5.61803, 5.90211}

betweenness = {1.92897, 19.7579, 32.9758, 41.7353, 46.102, 46.102, 41.7353, \
32.9758, 19.7579, 1.92897, 1.92897, 19.7579, 32.9758, 41.7353, \
46.102, 46.102, 41.7353, 32.9758, 19.7579, 1.92897}
```

B.5.3 Ladder 20 nodes results

```
avg−degree = 2.9

min−degree = 2.

sort−eigen = {−2.97766, −2.91115, −2.80194, −2.65248, −2.4661, −2.24698, −2., \
 −1.73068, −1.44504, −1.14946, −0.977662, −0.911146, −0.85054, \
 −0.801938, −0.652478, −0.554958, −0.466104, −0.269318, −0.24698, 0., \
0., 0.24698, 0.269318, 0.466104, 0.554958, 0.652478, 0.801938, \
0.85054, 0.911146, 0.977662, 1.14946, 1.44504, 1.73068, 2., 2.24698, \
2.4661, 2.65248, 2.80194, 2.91115, 2.97766}

sort−lapl−eigen = {0., 0.0246233, 0.097887, 0.217987, 0.381966, 0.585786, 0.824429, \
1.09202, 1.38197, 1.68713, 2., 2., 2.02462, 2.09789, 2.21799, \
2.31287, 2.38197, 2.58579, 2.61803, 2.82443, 2.90798, 3.09202, \
3.17557, 3.38197, 3.41421, 3.61803, 3.68713, 3.78201, 3.90211, \
3.97538, 4., 4.31287, 4.61803, 4.90798, 5.17557, 5.41421, 5.61803, \
5.78201, 5.90211, 5.97538}

betweenness = {2.59774, 41.1455, 75.1406, 104.747, 130.044, 151.079, 167.881, \
180.468, 188.853, 193.044, 193.044, 188.853, 180.468, 167.881, \
```

```
151.079, 130.044, 104.747, 75.1406, 41.1455, 2.59774, 2.59774, \
41.1455, 75.1406, 104.747, 130.044, 151.079, 167.881, 180.468, \
188.853, 193.044, 193.044, 188.853, 180.468, 167.881, 151.079, \
130.044, 104.747, 75.1406, 41.1455, 2.59774}
```

B.5.4 Ladder 40 nodes results

```
avg−degree = 2.95

min−degree = 2.

sort−eigen = {−2.99413, −2.97656, −2.94739, −2.90679, −2.855, −2.79233, −2.71914, \
−2.63586, −2.54298, −2.44104, −2.33065, −2.21245, −2.08714, −1.95544, \
−1.81814, −1.67603, −1.52996, −1.38078, −1.22937, −1.07661, \
−0.994132, −0.976561, −0.947391, −0.923395, −0.906793, −0.855005, \
−0.792331, −0.770633, −0.719139, −0.635859, −0.619218, −0.542978, \
−0.470037, −0.441043, −0.330651, −0.323966, −0.212451, −0.181863, \
−0.0871351, −0.0445604, 0.0445604, 0.0871351, 0.181863, 0.212451, \
0.323966, 0.330651, 0.441043, 0.470037, 0.542978, 0.619218, 0.635859, \
0.719139, 0.770633, 0.792331, 0.855005, 0.906793, 0.923395, 0.947391, \
0.976561, 0.994132, 1.07661, 1.22937, 1.38078, 1.52996, 1.67603, \
1.81814, 1.95544, 2.08714, 2.21245, 2.33065, 2.44104, 2.54298, \
2.63586, 2.71914, 2.79233, 2.855, 2.90679, 2.94739, 2.97656, 2.99413}

sort−lapl−eigen = {0., 0.00616533, 0.0246233, 0.0552602, 0.097887, 0.152241, 0.217987, \
0.29472, 0.381966, 0.479188, 0.585786, 0.701104, 0.824429, 0.955003, \
1.09202, 1.23463, 1.38197, 1.53311, 1.68713, 1.84308, 2., 2., \
2.00617, 2.02462, 2.05526, 2.09789, 2.15224, 2.15692, 2.21799, \
2.29472, 2.31287, 2.38197, 2.46689, 2.47919, 2.58579, 2.61803, \
2.7011, 2.76537, 2.82443, 2.90798, 2.955, 3.045, 3.09202, 3.17557, \
3.23463, 3.2989, 3.38197, 3.41421, 3.52081, 3.53311, 3.61803, \
3.68713, 3.70528, 3.78201, 3.84308, 3.84776, 3.90211, 3.94474, \
3.97538, 3.99383, 4., 4.15692, 4.31287, 4.46689, 4.61803, 4.76537, \
4.90798, 5.045, 5.17557, 5.2989, 5.41421, 5.52081, 5.61803, 5.70528, \
5.78201, 5.84776, 5.90211, 5.94474, 5.97538, 5.99383}

betweenness = {3.27854, 82.5321, 157.26, 227.628, 293.719, 355.583, 413.251, \
466.747, 516.088, 561.286, 602.352, 639.294, 672.118, 700.829, \
725.432, 745.93, 762.325, 774.62, 782.815, 786.913, 786.913, 782.815, \
774.62, 762.325, 745.93, 725.432, 700.829, 672.118, 639.294, 602.352, \
561.286, 516.088, 466.747, 413.251, 355.583, 293.719, 227.628, \
157.26, 82.5321, 3.27854, 3.27854, 82.5321, 157.26, 227.628, 293.719, \
355.583, 413.251, 466.747, 516.088, 561.286, 602.352, 639.294, \
672.118, 700.829, 725.432, 745.93, 762.325, 774.62, 782.815, 786.913, \
786.913, 782.815, 774.62, 762.325, 745.93, 725.432, 700.829, 672.118, \
639.294, 602.352, 561.286, 516.088, 466.747, 413.251, 355.583, \
293.719, 227.628, 157.26, 82.5321, 3.27854}
```

B.5.5 Ladder 80 nodes results

```
avg−degree = 2.975

min−degree = 2.

sort−eigen = {−2.9985, −2.99399, −2.98648, −2.97598, −2.96251, −2.94609, −2.92674, \
−2.9045, −2.87939, −2.85145, −2.82073, −2.78727, −2.75112, −2.71233, \
−2.67098, −2.6271, −2.58079, −2.53209, −2.48109, −2.42786, −2.37248, \
−2.31504, −2.25562, −2.19432, −2.13121, −2.06641, −2., −1.93209, \
−1.86277, −1.79216, −1.72036, −1.64747, −1.57361, −1.49888, −1.42341, \
−1.3473, −1.27066, −1.19362, −1.11629, −1.03878, −0.998496, \
```

```
−0.993986, −0.986477, −0.97598, −0.962511, −0.961217, −0.94609, \
−0.926742, −0.904496, −0.88371, −0.879385, −0.851448, −0.820726, \
−0.806378, −0.787265, −0.751116, −0.729337, −0.712334, −0.670976, \
−0.652704, −0.627104, −0.580785, −0.576592, −0.532089, −0.501118, \
−0.481088, −0.427859, −0.426394, −0.372483, −0.352532, −0.315043, \
−0.279645, −0.255624, −0.20784, −0.194317, −0.137228, −0.131214, \
−0.067913, −0.0664089, 0., 0., 0.0664089, 0.067913, 0.131214, \
0.137228, 0.194317, 0.20784, 0.255624, 0.279645, 0.315043, 0.352532, \
0.372483, 0.426394, 0.427859, 0.481088, 0.501118, 0.532089, 0.576592, \
0.580785, 0.627104, 0.652704, 0.670976, 0.712334, 0.729337, 0.751116, \
0.787265, 0.806378, 0.820726, 0.851448, 0.879385, 0.88371, 0.904496, \
0.926742, 0.94609, 0.961217, 0.962511, 0.97598, 0.986477, 0.993986, \
0.998496, 1.03878, 1.11629, 1.19362, 1.27066, 1.3473, 1.42341, \
1.49888, 1.57361, 1.64747, 1.72036, 1.79216, 1.86277, 1.93209, 2., \
2.06641, 2.13121, 2.19432, 2.25562, 2.31504, 2.37248, 2.42786, \
2.48109, 2.53209, 2.58079, 2.6271, 2.67098, 2.71233, 2.75112, \
2.78727, 2.82073, 2.85145, 2.87939, 2.9045, 2.92674, 2.94609, \
2.96251, 2.97598, 2.98648, 2.99399, 2.9985}

sort−lapl−eigen = {0., 0.00154193, 0.00616533, 0.0138631, 0.0246233, 0.0384294, \
0.0552602, 0.0750895, 0.097887, 0.123617, 0.152241, 0.183714, \
0.217987, 0.255008, 0.29472, 0.337061, 0.381966, 0.429366, 0.479188, \
0.531355, 0.585786, 0.642399, 0.701104, 0.761812, 0.824429, 0.88886, \
0.955003, 1.02276, 1.09202, 1.16268, 1.23463, 1.30777, 1.38197, \
1.45712, 1.53311, 1.60982, 1.68713, 1.76493, 1.84308, 1.92148, 2., \
2., 2.00154, 2.00617, 2.01386, 2.02462, 2.03843, 2.05526, 2.07509, \
2.07852, 2.09789, 2.12362, 2.15224, 2.15692, 2.18371, 2.21799, \
2.23507, 2.25501, 2.29472, 2.31287, 2.33706, 2.38197, 2.39018, \
2.42937, 2.46689, 2.47919, 2.53135, 2.54288, 2.58579, 2.61803, \
2.6424, 2.69223, 2.7011, 2.76181, 2.76537, 2.82443, 2.83732, 2.88886, \
2.90798, 2.955, 2.97724, 3.02276, 3.045, 3.09202, 3.11114, 3.16268, \
3.17557, 3.23463, 3.23819, 3.2989, 3.30777, 3.3576, 3.38197, 3.41421, \
3.45712, 3.46865, 3.52081, 3.53311, 3.57063, 3.60982, 3.61803, \
3.66294, 3.68713, 3.70528, 3.74499, 3.76493, 3.78201, 3.81629, \
3.84308, 3.84776, 3.87638, 3.90211, 3.92148, 3.92491, 3.94474, \
3.96157, 3.97538, 3.98614, 3.99383, 3.99846, 4., 4.07852, 4.15692, \
4.23507, 4.31287, 4.39018, 4.46689, 4.54288, 4.61803, 4.69223, \
4.76537, 4.83732, 4.90798, 4.97724, 5.045, 5.11114, 5.17557, 5.23819, \
5.2989, 5.3576, 5.41421, 5.46865, 5.52081, 5.57063, 5.61803, 5.66294, \
5.70528, 5.74499, 5.78201, 5.81629, 5.84776, 5.87638, 5.90211, \
5.92491, 5.94474, 5.96157, 5.97538, 5.98614, 5.99383, 5.99846}

betweenness = {3.96548, 163.918, 319.359, 470.453, 617.284, 759.902, 898.34, \
1032.62, 1162.76, 1288.78, 1410.69, 1528.49, 1642.19, 1751.8, \
1857.32, 1958.76, 2056.12, 2149.41, 2238.63, 2323.78, 2404.86, \
2481.88, 2554.83, 2623.73, 2688.56, 2749.34, 2806.06, 2858.73, \
2907.34, 2951.89, 2992.4, 3028.85, 3061.25, 3089.6, 3113.9, 3134.15, \
3150.35, 3162.49, 3170.59, 3174.64, 3174.64, 3170.59, 3162.49, \
3150.35, 3134.15, 3113.9, 3089.6, 3061.25, 3028.85, 2992.4, 2951.89, \
2907.34, 2858.73, 2806.06, 2749.34, 2688.56, 2623.73, 2554.83, \
2481.88, 2404.86, 2323.78, 2238.63, 2149.41, 2056.12, 1958.76, \
1857.32, 1751.8, 1642.19, 1528.49, 1410.69, 1288.78, 1162.76, \
1032.62, 898.34, 759.902, 617.284, 470.453, 319.359, 163.918, \
3.96548, 3.96548, 163.918, 319.359, 470.453, 617.284, 759.902, \
898.34, 1032.62, 1162.76, 1288.78, 1410.69, 1528.49, 1642.19, 1751.8, \
1857.32, 1958.76, 2056.12, 2149.41, 2238.63, 2323.78, 2404.86, \
2481.88, 2554.83, 2623.73, 2688.56, 2749.34, 2806.06, 2858.73, \
2907.34, 2951.89, 2992.4, 3028.85, 3061.25, 3089.6, 3113.9, 3134.15, \
3150.35, 3162.49, 3170.59, 3174.64, 3174.64, 3170.59, 3162.49, \
3150.35, 3134.15, 3113.9, 3089.6, 3061.25, 3028.85, 2992.4, 2951.89, \
2907.34, 2858.73, 2806.06, 2749.34, 2688.56, 2623.73, 2554.83, \
2481.88, 2404.86, 2323.78, 2238.63, 2149.41, 2056.12, 1958.76, \
1857.32, 1751.8, 1642.19, 1528.49, 1410.69, 1288.78, 1162.76, \
1032.62, 898.34, 759.902, 617.284, 470.453, 319.359, 163.918, 3.96548}
```

B.6 Powerlaw results

B.6.1 Powerlaw 10 nodes results

```
avg−degree = 4.6

min−degree = 3.

sort−eigen = {−2.67852, −2.28605, −1.98024, −0.683964, −0.361555, 0.177672, \
0.246201, 1.03157, 1.45485, 5.08003}

sort−lapl−eigen = {0., 2.11559, 2.38197, 3.4191, 4.12441, 4.61803, 5.20732, 6.85593, \
8.16729, 9.11037}

betweenness = {1.08333, 1.58333, 0.25, 0.25, 6.16667, 8.41667, 0.25, 1.5, 0.833333, \
1.66667}
```

B.6.2 Powerlaw 20 nodes results

```
avg−degree = 3.6

min−degree = 2.

sort−eigen = {−2.9878, −2.58877, −2.39235, −2.0337, −1.54575, −1.3155, −1.05624, \
−0.860999, −0.370421, −0.0814312, 0.0740858, 0.257016, 0.437427, \
0.894328, 1.09865, 1.35784, 1.54617, 2.17849, 2.73449, 4.65446}

sort−lapl−eigen = {0., 0.738646, 1.04335, 1.40346, 1.54101, 1.55536, 1.68942, 1.87163, \
2., 2.40914, 2.86903, 3.51113, 3.82977, 4.32659, 4.59192, 4.72652, \
6.1083, 6.63905, 8.97413, 12.1715}

betweenness = {0.833333, 0., 75.4702, 10.1012, 2.11905, 17.369, 14.5, 1.08333, \
51.2381, 20.8214, 0., 11.2262, 1.75, 6.34524, 2.60119, 0., 0.5, \
0.875, 0., 1.16667}
```

B.6.3 Powerlaw 40 nodes results

```
avg−degree = 7.1

min−degree = 3.

sort−eigen = {−4.10448, −3.76071, −3.40684, −3.3282, −3.00467, −2.59318, −2.45595, \
−2.41124, −2.19674, −2.16474, −2.07693, −1.89726, −1.7391, −1.39924, \
−1.18442, −0.912867, −0.781922, −0.70984, −0.527677, −0.45283, \
−0.390412, −0.0566637, 0.283756, 0.304728, 0.458534, 0.638499, \
0.733821, 0.927821, 1.12607, 1.24353, 1.77768, 2.02306, 2.24635, \
2.57659, 2.58452, 2.84525, 3.5204, 4.08585, 5.09013, 9.08932}

sort−lapl−eigen = {0., 2.00316, 2.16943, 2.49046, 2.55775, 2.70806, 2.92492, 3.03849, \
3.31842, 3.38519, 3.51128, 3.61094, 3.77009, 3.98445, 4.48245, \
4.88382, 5.01375, 5.32147, 5.43304, 5.80836, 5.94018, 6.09349, \
6.67248, 7.0542, 7.46469, 7.54008, 7.86685, 8.18826, 8.92477, \
9.04738, 9.48804, 9.89448, 10.1339, 11.4197, 12.9021, 13.6891, \
14.6518, 16.1636, 19.8942, 20.5551}

betweenness = {18.8383, 119.698, 1.22736, 31.6167, 77.032, 52.6332, 24.6981, \
60.7978, 131.332, 67.0504, 25.0663, 18.6372, 6.15946, 21.4835, \
```

```
8.23366, 16.7501, 2.26786, 9.37656, 18.762, 21.0812, 8.74295, 0., \
4.41111, 2.40952, 7.91639, 15.5313, 1.39803, 1.86627, 7.71105, \
9.98991, 5.94121, 2.93939, 1.90931, 1.94286, 5.32662, 0.627778, \
8.62249, 3.46285, 4.74526, 3.76349}
```

B.6.4 Powerlaw 80 nodes results

```
avg−degree = 7.575

min−degree = 2.

sort−eigen = {−5.54485, −4.54641, −4.11727, −4.04955, −3.69566, −3.6035, −3.42362, \
−3.18616, −3.01392, −2.92198, −2.89434, −2.8051, −2.68024, −2.44793, \
−2.41372, −2.17886, −2.16013, −2.11747, −1.9883, −1.88654, −1.80619, \
−1.76275, −1.67338, −1.6526, −1.58784, −1.45641, −1.3823, −1.29973, \
−1.21092, −1.14495, −1.04963, −0.960663, −0.923545, −0.857585, \
−0.754817, −0.674224, −0.576932, −0.529781, −0.349443, −0.308044, \
−0.272524, −0.120195, −0.0138591, 0.000335003, 0.123722, 0.185331, \
0.256086, 0.318917, 0.416397, 0.508203, 0.591087, 0.645816, 0.748248, \
0.893846, 0.966238, 1.08181, 1.11452, 1.1836, 1.25572, 1.44102, \
1.52286, 1.66874, 1.71066, 1.88477, 1.94053, 2.08667, 2.39568, \
2.44233, 2.60644, 2.78208, 3.01812, 3.33137, 3.42752, 3.77526, \
4.21185, 4.61132, 4.78521, 6.39382, 7.00198, 10.7158}

sort−lapl−eigen = {0., 1.72714, 1.76225, 1.9687, 2.03383, 2.24247, 2.34628, 2.54961, \
2.63221, 2.66859, 2.82572, 2.88915, 2.95961, 3.15753, 3.17927, \
3.35415, 3.40395, 3.4641, 3.50828, 3.56617, 3.59645, 3.6113, 3.642, \
3.73006, 3.83845, 4.08438, 4.1005, 4.2757, 4.45292, 4.47409, 4.87452, \
5.12232, 5.14517, 5.27122, 5.44008, 5.49748, 5.57972, 5.66118, \
5.79677, 6.00708, 6.17507, 6.30215, 6.37896, 6.43251, 6.52908, \
6.69309, 6.73091, 6.81052, 6.96395, 6.98281, 7.16375, 7.37259, \
7.61664, 7.83007, 7.96846, 8.01662, 8.09256, 8.52305, 8.65812, \
8.7532, 9.06769, 9.34863, 9.82789, 10.6714, 10.8444, 11.2844, \
11.8195, 12.2788, 12.4143, 13.2687, 13.4649, 14.0117, 15.5859, \
16.0625, 17.7357, 19.1758, 19.8204, 21.4071, 24.3884, 41.0874}

betweenness = {254.047, 44.5925, 199.216, 0., 389.481, 104.989, 20.181, 113.042, \
1039.85, 259.866, 31.3838, 70.3958, 50.7388, 3.189, 139.245, 19.8387, \
111.366, 14.3465, 72.6528, 200.162, 78.0933, 35.0301, 33.1498, \
7.93807, 14.8951, 13.2388, 111.809, 8.44466, 18.9217, 22.1952, \
49.7636, 0.77619, 29.1953, 14.3348, 14.8792, 22.9724, 41.1381, \
7.04057, 11.6376, 15.8112, 7.41175, 26.7649, 4.51574, 11.373, \
40.9899, 5.32495, 38.1317, 2.80229, 30.9638, 10.5511, 11.9145, \
31.5571, 12.19, 21.3533, 22.947, 6.55227, 4.30593, 12.7245, 7.70127, \
1.28301, 0.933333, 1.33889, 11.9752, 2.31602, 8.67804, 3.63526, \
4.97023, 18.4709, 7.97945, 14.3743, 2.04325, 7.92896, 3.90467, \
6.08004, 10.1608, 14.9071, 8.23526, 7.16175, 5.42417, 1.27332}
```

B.6.5 Powerlaw 160 nodes results

```
avg−degree = 7.725

min−degree = 3.

sort−eigen = {−6.61176, −5.95997, −5.63493, −5.22546, −5.16471, −4.70032, \
−4.47623, −4.26043, −4.1606, −3.78614, −3.60985, −3.40878, −3.26971, \
−3.19768, −3.17033, −3.11107, −3.06317, −2.97363, −2.91735, −2.78545, \
−2.76587, −2.73328, −2.58058, −2.54733, −2.48344, −2.42285, −2.33349, \
−2.3039, −2.25554, −2.22856, −2.16687, −2.13758, −2.05873, −2.01028, \
−1.97668, −1.94334, −1.92516, −1.87264, −1.81605, −1.79646, −1.74647, \
```

```
−1.71154, −1.6444, −1.6071, −1.58459, −1.51925, −1.50783, −1.47748, \
−1.45829, −1.41487, −1.37976, −1.31469, −1.27219, −1.26287, −1.19689, \
−1.18332, −1.13764, −1.04384, −1.03118, −1.01668, −0.97642, \
−0.950967, −0.949484, −0.863024, −0.853087, −0.823232, −0.755525, \
−0.735554, −0.694551, −0.680986, −0.622582, −0.528758, −0.513172, \
−0.482726, −0.42322, −0.400367, −0.390161, −0.332869, −0.302989, \
−0.214714, −0.194507, −0.164716, −0.146545, −0.121066, −0.0696091, \
−0.043826, −0.0377777, 0.0739101, 0.131358, 0.143127, 0.202618, \
0.221559, 0.255244, 0.293059, 0.352386, 0.375825, 0.435439, 0.440899, \
0.473868, 0.500932, 0.52023, 0.576511, 0.621926, 0.671032, 0.704074, \
0.736518, 0.831154, 0.863147, 0.944395, 0.964388, 0.97331, 1.02919, \
1.04827, 1.10265, 1.19678, 1.21378, 1.26816, 1.309, 1.34875, 1.38827, \
1.43003, 1.49932, 1.52129, 1.65486, 1.66903, 1.78401, 1.85423, \
1.93367, 1.95138, 2.05198, 2.11359, 2.1936, 2.20764, 2.36281, \
2.44578, 2.50719, 2.58171, 2.71011, 2.84347, 2.93947, 3.02368, \
3.11361, 3.19494, 3.28449, 3.36299, 3.44255, 3.64219, 3.98379, \
4.15564, 4.30843, 4.57675, 4.66669, 4.80589, 5.14836, 5.75005, \
6.07755, 6.30621, 6.98766, 8.39161, 12.9395}

sort−lapl−eigen = {0., 1.57959, 1.68715, 1.8087, 1.99785, 2.06346, 2.07471, 2.15839, \
2.2244, 2.39755, 2.40906, 2.4727, 2.49186, 2.54077, 2.60297, 2.63016, \
2.67416, 2.7373, 2.74746, 2.77071, 2.83189, 2.88432, 2.88949, \
2.96843, 2.98559, 3.02311, 3.10039, 3.10368, 3.1595, 3.18822, \
3.23032, 3.30624, 3.31618, 3.346, 3.35626, 3.38765, 3.4238, 3.46606, \
3.49222, 3.50239, 3.54435, 3.55695, 3.60528, 3.61729, 3.67118, \
3.68841, 3.71005, 3.73075, 3.7588, 3.77635, 3.79881, 3.81906, \
3.85874, 3.86309, 3.89652, 3.94148, 4.08484, 4.14675, 4.20559, \
4.23957, 4.26189, 4.34206, 4.35112, 4.39237, 4.51443, 4.52385, \
4.60167, 4.70311, 4.73497, 4.78549, 4.85146, 4.94748, 4.98187, \
5.02388, 5.18908, 5.23499, 5.32548, 5.40274, 5.4264, 5.51105, \
5.54292, 5.58431, 5.70044, 5.70498, 5.75109, 5.86594, 5.86926, \
6.05098, 6.19039, 6.2513, 6.27575, 6.32206, 6.48356, 6.54365, \
6.55511, 6.62153, 6.63053, 6.68527, 6.72133, 6.75006, 6.77882, \
6.89264, 6.94526, 6.97797, 7.0971, 7.11648, 7.25373, 7.30374, \
7.32594, 7.60119, 7.7748, 7.81186, 8.06309, 8.18658, 8.39944, \
8.45011, 8.51136, 8.53314, 8.61761, 8.66923, 8.80831, 8.82958, \
8.8385, 8.91523, 9.06183, 9.36784, 9.42975, 9.56541, 9.62682, \
9.79654, 9.94038, 10.0864, 10.1387, 10.3028, 10.5352, 10.6421, \
11.2284, 11.3702, 11.7214, 12.7504, 13.178, 13.423, 13.5175, 13.6782, \
13.999, 14.5509, 14.883, 15.7735, 16.3692, 18.2104, 19.663, 21.9441, \
26.1091, 28.97, 30.899, 31.022, 33.3232, 41.8069, 43.4925, 50.1712}

betweenness = {1131.96, 541.357, 1165.13, 111.571, 1936.81, 867.625, 197.247, \
337.36, 2569.68, 799.26, 142.596, 2000.82, 31.0446, 52.1817, 17.4557, \
40.13, 793.182, 221.486, 276.974, 596.715, 122.03, 71.3657, 32.4194, \
52.4741, 195.72, 105.152, 181.632, 415.409, 74.8097, 4.80755, \
91.1132, 112.261, 31.077, 277.019, 275.842, 76.5845, 46.8882, \
27.4938, 48.1736, 12.3674, 13.7149, 14.1733, 120.046, 21.8149, \
90.6981, 22.8788, 13.0361, 2.80913, 84.2024, 68.7652, 74.9284, \
129.331, 73.2838, 54.8071, 22.7048, 4.15865, 51.0139, 80.6759, \
288.401, 63.5261, 9.90238, 3.90439, 13.2829, 7.9572, 3.29162, \
53.2133, 11.0336, 76.1531, 57.299, 39.1467, 16.3497, 0., 51.542, \
6.9195, 63.7965, 58.8305, 41.5706, 59.3574, 1.92017, 11.068, 158.025, \
39.4878, 68.7956, 23.182, 17.0398, 22.0017, 33.0478, 16.7959, \
33.4992, 0.25, 64.0664, 39.213, 13.2249, 18.298, 64.7322, 8.84465, \
25.447, 8.9215, 26.7429, 9.10181, 3.8856, 67.8191, 25.3397, 6.33947, \
45.2908, 102.81, 59.066, 44.8804, 22.3845, 86.4538, 10.9404, 17.1474, \
6.59856, 64.964, 9.48722, 61.7909, 16.8481, 10.2268, 14.5149, \
9.05809, 10.7812, 8.74922, 2.95833, 5.94757, 7.11704, 4.99227, \
24.5592, 6.01746, 0., 9.95158, 3.1599, 16.5097, 3.04546, 3.56951, \
18.5141, 13.6729, 11.759, 86.829, 21.6493, 13.932, 22.6841, 1.20135, \
31.9049, 10.4054, 9.31177, 21.5512, 8.2248, 9.64006, 7.75374, 19.857, \
10.9526, 19.9045, 2.92642, 13.3188, 8.33793, 16.2209, 8.19998, \
1.0631, 7.33123, 6.20149}
```

B.7 Regular results

B.7.1 Regular 10 nodes results

```
avg−degree = 4.

min−degree = 4.

sort−eigen = {−2.65544, −2.21432, −2., −1.21076, 0., 0., 0.539189, 1.67513, \
1.8662, 4.}

sort−lapl−eigen = {0., 2.1338, 2.32487, 3.46081, 4., 4., 5.21076, 6., 6.21432, 6.65544}

betweenness = {2.33333, 2.5, 2.33333, 2.33333, 3., 2.5, 2.33333, 2.33333, 2.33333, \
3.}
```

B.7.2 Regular 20 nodes results

```
avg−degree = 4.

min−degree = 4.

sort−eigen = {−3.23632, −2.81669, −2.49394, −1.91249, −1.76826, −1.52319, \
−1.20473, −1.13761, −0.606473, −0.166174, −0.0442904, 0.176211, \
0.742367, 0.868527, 1.7413, 1.80748, 2.05683, 2.62091, 2.89654, 4.}

sort−lapl−eigen = {0., 1.10346, 1.37909, 1.94317, 2.19252, 2.2587, 3.13147, 3.25763, \
3.82379, 4.04429, 4.16617, 4.60647, 5.13761, 5.20473, 5.52319, \
5.76826, 5.91249, 6.49394, 6.81669, 7.23632}

betweenness = {7.63889, 9.62778, 17.1278, 11.15, 7.79444, 13.4389, 7.76667, \
11.2167, 15.5333, 15.9222, 9.16111, 12.5667, 11.7278, 7.17222, \
10.0833, 9.83889, 9.77778, 8.52222, 14.9333, 11.}
```

B.7.3 Regular 40 nodes results

```
avg−degree = 4.

min−degree = 4.

sort−eigen = {−3.32099, −3.15683, −3.11369, −2.8707, −2.60133, −2.50256, −2.24191, \
−2.11072, −1.86519, −1.70886, −1.56854, −1.44887, −1.33037, −1.16456, \
−1.01491, −0.9146, −0.649398, −0.537933, −0.235115, −0.115034, \
0.12512, 0.323524, 0.375424, 0.525161, 0.591361, 0.816921, 1.17682, \
1.28305, 1.45136, 1.69921, 1.80085, 1.87163, 2.04754, 2.23282, \
2.45078, 2.70703, 2.85948, 3.04902, 3.08499, 4.}

sort−lapl−eigen = {0., 0.915015, 0.950979, 1.14052, 1.29297, 1.54922, 1.76718, 1.95246, \
2.12837, 2.19915, 2.30079, 2.54864, 2.71695, 2.82318, 3.18308, \
3.40864, 3.47484, 3.62458, 3.67648, 3.87488, 4.11503, 4.23512, \
4.53793, 4.6494, 4.9146, 5.01491, 5.16456, 5.33037, 5.44887, 5.56854, \
5.70886, 5.86519, 6.11072, 6.24191, 6.50256, 6.60133, 6.8707, \
7.11369, 7.15683, 7.32099}

betweenness = {18.3001, 36.131, 45.3579, 38.2389, 41.9359, 44.1937, 30.196, \
30.3056, 22.0587, 32.5833, 31.9179, 40.123, 30.8452, 30.7211, \
30.5775, 29.8262, 24.4028, 40.21, 36.0298, 23.2885, 34.7302, 34.0104, \
```

```
36.5833, 36.6782, 26.3242, 31.1092, 33.6204, 35.9274, 41.6151, \
35.4977, 31.9358, 38.2235, 22.5, 41.6151, 49.2997, 40.3179, 29.508, \
37.8485, 36.514, 35.8985}
```

B.7.4 Regular 80 nodes results

```
avg−degree = 4.

min−degree = 4.

sort−eigen = {−3.3307, −3.26477, −3.17404, −2.97568, −2.93956, −2.89891, −2.80148, \
−2.731, −2.67349, −2.59099, −2.48873, −2.44916, −2.35492, −2.31844, \
−2.20653, −2.09415, −2.03676, −2.00393, −1.93272, −1.80551, −1.72365, \
−1.63513, −1.51068, −1.48916, −1.39477, −1.27252, −1.25028, −1.21275, \
−1.03382, −0.958459, −0.924623, −0.882749, −0.679355, −0.547896, \
−0.460763, −0.42179, −0.371972, −0.252392, −0.101341, −0.0576326, \
0.00942982, 0.171013, 0.198652, 0.255303, 0.396111, 0.425986, \
0.500499, 0.530451, 0.645057, 0.806362, 0.876255, 0.936051, 1.01216, \
1.08929, 1.1855, 1.28357, 1.47579, 1.54013, 1.56399, 1.66642, \
1.73445, 1.77082, 1.96313, 2.00288, 2.11481, 2.19026, 2.3511, \
2.48118, 2.60082, 2.63736, 2.69441, 2.79584, 2.80603, 2.84021, \
2.94677, 3.06194, 3.15848, 3.19693, 3.33773, 4.}

sort−lapl−eigen = {0., 0.662269, 0.80307, 0.841521, 0.938057, 1.05323, 1.15979, \
1.19397, 1.20416, 1.30559, 1.36264, 1.39918, 1.51882, 1.6489, \
1.80974, 1.88519, 1.99712, 2.03687, 2.22918, 2.26555, 2.33358, \
2.43601, 2.45987, 2.52421, 2.71643, 2.8145, 2.91071, 2.98784, \
3.06395, 3.12375, 3.19364, 3.35494, 3.46955, 3.4995, 3.57401, \
3.60389, 3.7447, 3.80135, 3.82899, 3.99057, 4.05763, 4.10134, \
4.25239, 4.37197, 4.42179, 4.46076, 4.5479, 4.67936, 4.88275, \
4.92462, 4.95846, 5.03382, 5.21275, 5.25028, 5.27252, 5.39477, \
5.48916, 5.51068, 5.63513, 5.72365, 5.80551, 5.93272, 6.00393, \
6.03676, 6.09415, 6.20653, 6.31844, 6.35492, 6.44916, 6.48873, \
6.59099, 6.67349, 6.731, 6.80148, 6.89891, 6.93956, 6.97568, 7.17404, \
7.26477, 7.3307}

betweenness = {106.232, 98.2067, 104.689, 102.544, 136.637, 66.4933, 78.6738, \
85.061, 103.983, 82.3262, 120.677, 97.0954, 114.241, 96.9691, \
102.516, 103.102, 98.1837, 101.706, 85.9896, 95.8696, 67.5464, \
82.8611, 103.914, 78.9206, 100.966, 123.042, 108.264, 79.5278, \
113.661, 95.5754, 92.6723, 96.6859, 78.1889, 71.6552, 72.7611, \
93.3808, 84.1481, 69.5205, 101.899, 87.1443, 59.5668, 97.3263, \
100.299, 113.231, 90.5798, 96.2921, 118.17, 100.795, 112.275, \
109.382, 72.622, 100.01, 90.519, 139.129, 86.5925, 90.4076, 86.2901, \
79.5026, 72.8405, 107.012, 91.6452, 129.456, 52.6795, 55.8926, \
107.109, 78.194, 79.3704, 100.552, 99.0236, 83.2508, 77.1493, \
79.3629, 120.603, 76.0226, 75.3964, 87.8787, 143.79, 91.3456, 95.713, \
78.1915}
```

B.7.5 Regular 160 nodes results

```
avg−degree = 4.

min−degree = 4.

sort−eigen = {−3.41861, −3.31216, −3.2688, −3.23909, −3.22421, −3.14629, −3.11413, \
−3.06368, −3.05764, −3.02004, −2.95671, −2.93548, −2.88392, −2.85379, \
−2.83838, −2.76258, −2.71437, −2.69412, −2.67407, −2.61999, −2.60514, \
−2.52688, −2.47293, −2.45811, −2.39097, −2.36236, −2.28975, −2.24135, \
−2.22708, −2.20637, −2.14719, −2.1109, −2.02546, −2.00301, −1.97022, \
```

```
−1.94938, −1.89363, −1.85873, −1.81193, −1.77397, −1.76356, −1.67328, \
−1.62125, −1.60907, −1.55188, −1.52294, −1.49663, −1.4891, −1.41181, \
−1.3396, −1.31638, −1.24913, −1.22926, −1.19397, −1.17559, −1.08914, \
−1.05813, −1.01043, −0.972393, −0.951802, −0.922095, −0.879528, \
−0.8203, −0.77974, −0.740264, −0.69271, −0.61896, −0.583418, \
−0.522146, −0.48546, −0.478487, −0.405435, −0.338481, −0.30377, \
−0.270975, −0.183694, −0.166768, −0.125212, −0.0473816, −0.00796921, \
0.0269374, 0.0764749, 0.1143, 0.129438, 0.1755, 0.240852, 0.313017, \
0.340149, 0.368985, 0.392416, 0.466598, 0.471291, 0.548524, 0.589517, \
0.645602, 0.685842, 0.706922, 0.752592, 0.81697, 0.872903, 0.921431, \
0.991985, 1.02681, 1.06365, 1.08437, 1.13875, 1.16207, 1.21172, \
1.25977, 1.29409, 1.30542, 1.33725, 1.4198, 1.48228, 1.5344, 1.5763, \
1.64816, 1.65367, 1.67297, 1.75115, 1.77392, 1.82539, 1.84955, \
1.88476, 1.90607, 1.97218, 1.9957, 2.03915, 2.10266, 2.15996, \
2.19421, 2.22986, 2.28697, 2.33213, 2.38127, 2.39999, 2.45957, \
2.49646, 2.50153, 2.58495, 2.6464, 2.66549, 2.68895, 2.73409, \
2.77828, 2.80316, 2.8327, 2.89906, 2.92276, 2.94525, 2.97516, \
3.02426, 3.08614, 3.16193, 3.18606, 3.19725, 3.2477, 3.35121, \
3.42854, 4.}

sort−lapl−eigen = {0., 0.662269, 0.80307, 0.841521, 0.938057, 1.05323, 1.15979, \
1.19397, 1.20416, 1.30559, 1.36264, 1.39918, 1.51882, 1.6489, \
1.80974, 1.88519, 1.99712, 2.03687, 2.22918, 2.26555, 2.33358, \
2.43601, 2.45987, 2.52421, 2.71643, 2.8145, 2.91071, 2.98784, \
3.06395, 3.12375, 3.19364, 3.35494, 3.46955, 3.4995, 3.57401, \
3.60389, 3.7447, 3.80135, 3.82899, 3.99057, 4.05763, 4.10134, \
4.25239, 4.37197, 4.42179, 4.46076, 4.5479, 4.67936, 4.88275, \
4.92462, 4.95846, 5.03382, 5.21275, 5.25028, 5.27252, 5.39477, \
5.48916, 5.51068, 5.63513, 5.72365, 5.80551, 5.93272, 6.00393, \
6.03676, 6.09415, 6.20653, 6.31844, 6.35492, 6.44916, 6.48873, \
6.59099, 6.67349, 6.731, 6.80148, 6.89891, 6.93956, 6.97568, 7.17404, \
7.26477, 7.3307}

betweenness = {308.072, 220.119, 235.654, 275.123, 231.024, 216.58, 292.912, \
300.895, 175.932, 157.726, 298.45, 217.461, 210.317, 230.282, 261.64, \
241.862, 234.875, 241.44, 279.783, 161.799, 223.298, 278.647, 262.98, \
198.1, 264.01, 210.699, 230.104, 320.65, 212.486, 203.214, 245.726, \
218.931, 264.939, 225.847, 309.113, 244.465, 231.724, 190.626, \
252.079, 155.702, 158.189, 171.924, 240.772, 180.094, 294.189, \
268.693, 219.982, 235.063, 274.023, 278.037, 278.963, 263.13, \
122.663, 270.83, 260.55, 202.534, 187.993, 235.466, 244.596, 262.633, \
259.798, 161.224, 239.114, 131.136, 198.142, 240.337, 197.359, \
201.814, 212.25, 192.815, 259.624, 192.997, 267.437, 174.606, \
232.438, 173.091, 257.41, 271.746, 259.309, 245.213, 250.061, \
266.417, 191.317, 256.004, 244.71, 203.436, 283.806, 230.889, \
254.323, 156.903, 286.753, 223.527, 183.204, 261.384, 191.009, \
247.068, 207.383, 138.938, 289.808, 271.865, 246.822, 230.527, \
136.355, 256.616, 297.392, 250.375, 274.628, 244.608, 240.853, \
189.966, 191.859, 305.594, 245.569, 254.421, 244.775, 261.25, \
272.029, 234.882, 264.055, 182.999, 215.303, 233.256, 244.817, 259.5, \
272.998, 265.948, 235.325, 186.46, 232.344, 278.524, 279.148, \
211.461, 298.673, 193.107, 252.251, 288.587, 230.107, 272.417, \
193.497, 203.345, 253.956, 267.338, 239.202, 280.823, 234.523, \
208.335, 240.267, 231.611, 265.262, 184.008, 305.883, 233.461, \
189.713, 206.969, 312.064, 245.586, 299.499, 202.589, 271.837, \
297.156}
```

B.8 Star results

B.8.1 Star 10 nodes results

```
avg−degree = 1.81818

min−degree = 1.

sort−eigen = {−3.16228, 0., 0., 0., 0., 0., 0., 0., 0., 3.16228}

sort−lapl−eigen = {0., 1., 1., 1., 1., 1., 1., 1., 1., 11.}

betweenness = {45., 0., 0., 0., 0., 0., 0., 0., 0., 0.}
```

B.8.2 Star 20 nodes results

```
avg−degree = 1.90476

min−degree = 1.

sort−eigen = {−4.47214, 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., \
0., 0., 0., 0., 0., 4.47214}

sort−lapl−eigen = {0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., \
1., 1., 1., 21.}

betweenness = {190., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., \
0., 0., 0., 0., 0.}
```

B.8.3 Star 40 nodes results

```
avg−degree = 1.95122

min−degree = 1.

sort−eigen = {−6.32456, 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., \
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., \
0., 0., 0., 0., 0., 0., 0., 6.32456}

sort−lapl−eigen = {0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., \
1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., \
1., 1., 1., 1., 1., 1., 41.}

betweenness = {780., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., \
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., \
0., 0., 0., 0., 0., 0., 0., 0.}
```

B.8.4 Star 80 nodes results

```
avg−degree = 1.97531

min−degree = 1.

sort−eigen = {−8.94427, 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., \
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., \
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., \
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., \
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., \
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 8.94427}

sort−lapl−eigen = {0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., \
```

```
1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., \
1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., \
1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., \
1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 81.}

betweenness = {3160., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., \
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., \
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., \
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., \
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.}
```

B.8.5 Star 160 nodes results

```
avg−degree = 1.98758

min−degree = 1.

sort−eigen = {−12.6491, 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., \
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., \
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., \
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., \
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., \
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., \
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., \
0., 0., 0., 0., 0., 0., 0., 0., 12.6491}

sort−lapl−eigen = {0., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., \
1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., \
1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., \
1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., \
1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., \
1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., \
1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., \
1., 1., 1., 1., 1., 1., 161.}

betweenness = {12720., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., \
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., \
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., \
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., \
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., \
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., \
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., \
0., 0., 0., 0., 0., 0., 0., 0.}
```

B.9 Watts results

B.9.1 Watts 10 nodes results

```
avg−degree = 4.

min−degree = 2.
```

```
sort−eigen = {−2.67232, −2.09492, −1.69596, −0.894509, −0.448388, −0.296041, \
0.772834, 1.29418, 1.59957, 4.43555}

sort−lapl−eigen = {0., 1.4764, 1.95455, 2.6576, 3.45506, 4.03812, 5.14408, 5.77875, \
7.12184, 8.3736}

betweenness = {0.392857, 1.11905, 0., 11.8214, 1.08333, 3.20238, 5.59524, 1.16667, \
0.47619, 2.14286}
```

B.9.2 Watts 20 nodes results

```
avg−degree = 4.

min−degree = 2.

sort−eigen = {−3.06607, −2.85447, −2.64591, −2.33187, −1.79067, −1.43113, \
−1.01917, −0.847805, −0.586865, −0.109001, 0.0781524, 0.346795, \
0.702005, 1.12394, 1.25562, 1.82146, 2.04312, 2.28394, 2.57028, \
4.45764}

sort−lapl−eigen = {0., 1.0826, 1.21198, 1.436, 1.97255, 2.11911, 2.28678, 2.7648, \
3.27565, 3.6236, 3.91346, 4.47837, 4.6326, 4.87805, 5.54466, 6.23037, \
7.06152, 7.53941, 7.74123, 8.20726}

betweenness = {0.833333, 10.3429, 1.4, 17.1833, 12.6976, 4.97857, 20.4667, 1.66667, \
14.2333, 4.33333, 13.8333, 12.119, 17.3929, 11.269, 11.5357, 1.37619, \
25.9024, 18.519, 2., 9.91667}
```

B.9.3 Watts 40 nodes results

```
avg−degree = 4.

min−degree = 2.

sort−eigen = {−3.53101, −3.10802, −2.89068, −2.59663, −2.55422, −2.44513, \
−2.25471, −2.10656, −1.97419, −1.81886, −1.62297, −1.42891, −1.18053, \
−1.1379, −0.879707, −0.764165, −0.620582, −0.486088, −0.379913, \
−0.172711, −0.0255148, 0.343225, 0.378979, 0.521451, 0.614413, \
0.962118, 1.04133, 1.1068, 1.31312, 1.42743, 1.68337, 1.89647, \
1.94983, 2.09278, 2.29733, 2.59253, 2.67563, 3.0978, 3.33027, 4.65413}

sort−lapl−eigen = {0., 0.764938, 0.892709, 1.0317, 1.19751, 1.28065, 1.46704, 1.54975, \
1.68522, 1.82571, 2.05399, 2.13516, 2.3286, 2.56596, 2.65151, \
2.76349, 2.92298, 3.2415, 3.29175, 3.70909, 3.8947, 3.94506, 4.37791, \
4.47725, 4.58629, 4.79307, 4.94957, 5.44756, 5.59749, 5.7628, \
5.98594, 6.22187, 6.46749, 6.8062, 6.93537, 7.19285, 7.27829, 7.889, \
8.10231, 9.92972}

betweenness = {5.16667, 41.4504, 18.3381, 35.473, 44.7802, 62.329, 11.9992, \
26.5683, 35.9635, 42.6532, 13.969, 6.78333, 23.1095, 66.0611, \
19.4595, 18.2635, 42.7579, 15.5476, 108.675, 47.65, 59.1226, 17.7214, \
43.0417, 12.904, 37.3802, 23.827, 18.1024, 21.9857, 41.4238, 51.2921, \
43.0492, 17.627, 2.81667, 37.4345, 21.9655, 103.698, 17.05, 9.15952, \
47.1429, 28.2587}
```

B.9.4 Watts 80 nodes results

```
avg−degree = 4.

min−degree = 2.

sort−eigen = {−3.73895, −3.69688, −3.35554, −3.3154, −3.1105, −3.01722, −2.88171, \
−2.75433, −2.63159, −2.5964, −2.55443, −2.39397, −2.31869, −2.21705, \
−2.08098, −2.04844, −1.96358, −1.82381, −1.71081, −1.6893, −1.54161, \
−1.4859, −1.36489, −1.31769, −1.22279, −1.14104, −1.02006, −0.852108, \
−0.840293, −0.796404, −0.709883, −0.695182, −0.629338, −0.454006, \
−0.349987, −0.307293, −0.274592, −0.264204, −0.185317, −0.0995123, \
−0.0228168, 0.0882836, 0.172076, 0.22754, 0.276727, 0.4573, 0.486744, \
0.583259, 0.612835, 0.737775, 0.782544, 0.863908, 0.976564, 1.05878, \
1.14522, 1.15294, 1.25777, 1.3554, 1.41294, 1.50561, 1.64306, \
1.79388, 1.82041, 1.92466, 2.06672, 2.15702, 2.24193, 2.2959, 2.401, \
2.49065, 2.6067, 2.69131, 2.73042, 2.77224, 2.95676, 3.08061, \
3.26565, 3.37244, 3.57399, 4.43492}

sort−lapl−eigen = {0., 0.59332, 0.680571, 0.769221, 0.847823, 0.915597, 0.99648, \
1.10926, 1.18098, 1.26486, 1.28971, 1.35573, 1.40335, 1.4217, \
1.52631, 1.59172, 1.63656, 1.68185, 1.78008, 1.88175, 1.95167, \
2.06111, 2.12439, 2.212, 2.36664, 2.41913, 2.47173, 2.56076, 2.62294, \
2.71228, 2.87702, 2.88917, 3.05827, 3.15877, 3.30239, 3.3315, \
3.34569, 3.4201, 3.55931, 3.72912, 3.78607, 3.81371, 3.9578, 4.02873, \
4.13014, 4.26978, 4.42865, 4.49278, 4.5304, 4.59929, 4.68665, \
4.80501, 4.91319, 5.09367, 5.17354, 5.3905, 5.56493, 5.59231, 5.7045, \
5.89341, 6.06214, 6.12169, 6.16327, 6.28607, 6.38445, 6.58715, \
6.6498, 6.70887, 6.86917, 7.03921, 7.08138, 7.37497, 7.58529, \
7.69409, 7.86505, 8.01167, 8.28346, 8.56155, 8.8159, 8.89889}

betweenness = {85.8422, 70.3175, 121.616, 175.429, 82.6016, 60.9203, 74.4037, \
114.098, 137.512, 104.184, 49.6694, 207.751, 150.399, 209.928, \
212.377, 50.1365, 172.623, 87.1695, 7.64444, 101.059, 22.2025, \
257.911, 66.1889, 43.7671, 83.1831, 15.8873, 68.546, 86.1099, \
124.536, 49.9448, 84.2437, 149.609, 101.496, 32.3025, 73.9216, \
136.045, 144.919, 114.756, 6.48611, 191.069, 129.752, 53.1349, \
5.76918, 124.789, 88.02, 54.0845, 63.2357, 86.6361, 44.0565, 172.322, \
134.402, 44.8622, 55.3447, 21.1229, 28.704, 86.5274, 112.961, \
151.535, 141.583, 125.033, 57.2507, 57.3544, 85.5524, 145.612, \
23.4167, 77.3016, 185.59, 20.4087, 55.8698, 26.217, 39.3757, 214.083, \
130.025, 63.7094, 27.5678, 16.054, 20.567, 12.3667, 37.9623, 151.036}
```

B.9.5 Watts 160 nodes results

```
avg−degree = 4.

min−degree = 2.

sort−eigen = {−3.80141, −3.65738, −3.65456, −3.5255, −3.50846, −3.39537, −3.32676, \
−3.20477, −3.19284, −3.07775, −3.05172, −3.01746, −2.96913, −2.89276, \
−2.83942, −2.78894, −2.76366, −2.66303, −2.62207, −2.58077, −2.53352, \
−2.38913, −2.37663, −2.33871, −2.28791, −2.24305, −2.1817, −2.1383, \
−2.10976, −2.01629, −2.00545, −1.95215, −1.91991, −1.88546, −1.84769, \
−1.7383, −1.72689, −1.71723, −1.65044, −1.64518, −1.56866, −1.50422, \
−1.48855, −1.44579, −1.39516, −1.36236, −1.32836, −1.28579, −1.23615, \
−1.2147, −1.1999, −1.1242, −1.08339, −1.0602, −1.00461, −0.937352, \
−0.921732, −0.872765, −0.814686, −0.790071, −0.693383, −0.674268, \
−0.661219, −0.621326, −0.571322, −0.552898, −0.482174, −0.432031, \
−0.414927, −0.380676, −0.36604, −0.324676, −0.30065, −0.262889, \
−0.199914, −0.191, −0.140671, −0.123694, −0.0708665, −0.0505582, \
−0.0438517, 0.0334376, 0.0522745, 0.0793978, 0.152192, 0.18445, \
0.217963, 0.26362, 0.289898, 0.321503, 0.343531, 0.379626, 0.427139, \
```

```
0.428432, 0.46281, 0.512338, 0.565427, 0.661531, 0.701333, 0.712458, \
0.770313, 0.834485, 0.839052, 0.88961, 0.919315, 0.965454, 1.01319, \
1.03163, 1.08222, 1.10609, 1.14383, 1.17375, 1.22804, 1.26393, \
1.32897, 1.35489, 1.36056, 1.3947, 1.45287, 1.55341, 1.57518, \
1.64804, 1.65675, 1.69217, 1.70715, 1.76149, 1.88903, 1.95842, \
2.00413, 2.07628, 2.10858, 2.14515, 2.17531, 2.20432, 2.26572, \
2.34288, 2.36986, 2.4289, 2.45373, 2.49942, 2.58988, 2.59729, \
2.66977, 2.71277, 2.79042, 2.82031, 2.91694, 2.94893, 3.01965, \
3.14158, 3.17047, 3.20349, 3.30213, 3.35815, 3.4664, 3.56139, \
3.62834, 3.68599, 3.81957, 4.57345}

sort−lapl−eigen = {0., 0.562728, 0.603034, 0.619289, 0.677177, 0.730105, 0.752786, \
0.807291, 0.856309, 0.870066, 0.89004, 0.954886, 0.957121, 0.991875, \
1.01733, 1.05502, 1.07685, 1.09566, 1.11913, 1.17113, 1.20181, \
1.20533, 1.2333, 1.27445, 1.31352, 1.35515, 1.37666, 1.38247, \
1.41564, 1.50367, 1.53529, 1.55655, 1.56283, 1.60789, 1.65169, \
1.71715, 1.73548, 1.77173, 1.80578, 1.83408, 1.88469, 1.9154, \
1.92716, 1.97811, 2.00504, 2.08176, 2.11524, 2.16278, 2.17689, \
2.21784, 2.22934, 2.27042, 2.33717, 2.37715, 2.4141, 2.42739, \
2.50339, 2.56607, 2.56889, 2.59789, 2.69552, 2.73083, 2.77537, \
2.81657, 2.8459, 2.8927, 2.95715, 3.07156, 3.08582, 3.14299, 3.19182, \
3.21156, 3.22905, 3.29277, 3.3584, 3.4164, 3.4642, 3.48469, 3.49149, \
3.56207, 3.62444, 3.64176, 3.64764, 3.71826, 3.77144, 3.87755, \
3.90207, 3.94851, 4.00216, 4.07701, 4.15442, 4.21359, 4.25476, \
4.26779, 4.3841, 4.44164, 4.48061, 4.55599, 4.57848, 4.62258, \
4.66361, 4.68485, 4.72507, 4.829, 4.87207, 4.93556, 4.95655, 5.00938, \
5.05533, 5.10621, 5.2131, 5.30505, 5.32536, 5.38196, 5.47653, \
5.57441, 5.63364, 5.67005, 5.75219, 5.80819, 5.83935, 5.95249, \
5.96061, 6.09435, 6.12571, 6.22015, 6.27882, 6.3184, 6.42162, \
6.45549, 6.49765, 6.55355, 6.71127, 6.80366, 6.82825, 6.94956, \
6.97256, 7.08419, 7.12216, 7.23612, 7.35546, 7.44618, 7.60988, \
7.64058, 7.83596, 7.87783, 7.89265, 7.95182, 8.0978, 8.17002, \
8.29169, 8.39148, 8.46071, 8.73759, 8.9763, 9.02852, 9.24058, \
9.68692, 9.74392, 10.6734}

betweenness = {325.557, 294.463, 499.603, 192.878, 237.529, 361.536, 45.2136, \
70.6686, 279.742, 299.211, 321.12, 568.862, 311.22, 122.125, 94.3449, \
373.519, 218.327, 124.881, 73.1469, 82.7131, 180.636, 115.184, \
231.949, 12.4719, 422.605, 225.086, 24.9234, 356.727, 128.545, \
38.5659, 137.291, 249.572, 163.775, 566.075, 123.295, 174.806, \
184.587, 120.705, 133.391, 126.018, 36.1834, 122.479, 573.982, \
213.851, 176.96, 677.97, 423.677, 177.704, 41.7473, 133.861, 211.786, \
524.363, 108.108, 28.2527, 370.266, 384.839, 159.028, 123.497, \
311.652, 195.238, 108.721, 136.277, 205.157, 326.927, 193.654, \
176.854, 178.56, 105.001, 97.4209, 46.141, 164.287, 45.0945, 182.979, \
22.819, 89.4544, 395.988, 181.602, 345.461, 81.8699, 61.6888, \
65.8521, 28.9659, 108.201, 114.019, 21.2322, 87.098, 266.902, \
394.953, 131.734, 57.0663, 121.146, 653.26, 342.987, 282.16, 215.079, \
221.071, 266.135, 346.84, 29.964, 450.041, 265.293, 288.34, 260.751, \
233.341, 122.743, 77.5245, 574.314, 262.558, 57.0068, 267.964, \
396.985, 235.787, 1036.83, 364.906, 257.791, 31.4684, 61.1942, \
471.749, 499.064, 215.268, 15.1552, 347.766, 37.4214, 218.372, \
82.3917, 174.465, 167.277, 405.156, 80.4522, 233.607, 106.983, \
201.297, 69.6185, 101.362, 321.653, 159.806, 416.689, 366.425, \
159.672, 602.558, 322.324, 94.0582, 39.6724, 273.673, 13.9604, \
289.229, 415.7, 167.099, 120.833, 195.347, 174.375, 124.619, 93.7291, \
119.664, 370.017, 718.604, 34.1555, 791.823, 45.8577, 463.277}
```